ESSENTIALS

Matrix/Vector Basics

Multiplication: $C = AB \Leftrightarrow c_{ik} = \sum_{j=1}^{m} a_{ij} b_{jk}$

Orthogonal Matrix: (full rank, square matrix with orthonormal columns) $A^{-1} = A^T$, $AA^T = A^TA = I$, $\det(A) \in \{-1, 1\}$, $\det(A^TA) = 1$. **preserves:** inner product, norm, distance, angle, rank, matrix orthogonality

Inner product: $\langle x, y \rangle = x^Ty = \sum_{i=1}^{n} x_i y_i$.

$\langle x \pm y, x \pm y \rangle = \langle x, x \rangle \pm 2\langle x, y \rangle + \langle y, y \rangle$

$\langle x, y+z \rangle = \langle x, z \rangle + \langle x, y \rangle$. $\langle x, y \rangle = \|x\|_2 \|y\|_2 \cos(\theta)$

Outer product: $uv^T \Leftrightarrow (uv^T)_{ij} = u_i v_j$

Norms: $\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{\langle x, x \rangle}$, $\|M\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} m_{ij}^2}$

Probability / Statistics Basics

$P(x) = \sum_{y \in Y} P(x, y)$ $\quad P(x, y) = P(x|y)\,P(y)$

$\forall y \in Y: \sum_{x \in X} P(x|y) = 1$

$P(x|y) = \frac{P(x,y)}{P(y)} = \frac{P(y|x) \cdot P(x)}{P(y)}$ if $P(y) > 0$

posterior $p(A|B) = \frac{\text{prior } p(A) \cdot \text{likelihood } p(B|A)}{\text{evidence } p(B)}$

If i.i.d: $P(x_1, \dots, x_n) = \prod_{i=1}^{n} P(x_i)$

If X & Y independent: $P(x|y) = P(x) \Leftrightarrow P(y|x) = P(y)$

Expectation: $\mathbb{E}[f(X)] = \sum_{x \in X} P(x) f(x) = \int P(x) f(x)\, dx$

Variance: $\mathrm{Var}[X] = \mathbb{E}[(X - \mu_X)^2] = \sum_{x \in X} (x - \mu_X)^2 P(x) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$

Expectation Properties

Linearity: $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$, $\mathbb{E}[aX] = a\mathbb{E}[X]$

For constant: $X = c$ then $\mathbb{E}[X] = c$, so we can take the expectation of unrelated stuff and get the same: $\log p_\theta(x^{(i)}) = \mathbb{E}_{z \sim q_\phi(z|x^{(i)})}[\log p_\theta(x^{(i)})]$

Double: For any X: $\mathbb{E}[\mathbb{E}[X]] = \mathbb{E}[X]$.

Convex Function

$\forall x_1, x_2 \in X, \forall t \in [0,1]$: (also if $\forall x\ f''(x) \geq 0$)

$f(tx_1 + (1-t)x_2) \leq t f(x_1) + (1-t) f(x_2)$

Example: Show that if f is convex, any local optimum is global. Assume there is a local optimum $\hat{x}$ that is not the global optimum x^*, then if we choose t in the ball of the local optimum we know that: $f(t\hat{x} + (1-t)x^*) \geq f(\hat{x})$. Since $f(x^*) < f(\hat{x})$, we have $t \cdot f(\hat{x}) + (1-t) f^*(x) < f(\hat{x})$. So we get $f(t\hat{x} + (1-t)x^*) \geq f(\hat{x}) > t \cdot f(\hat{x}) + (1-t) f^*(x)$ which contradicts the convexity of f.

Jensen Inequality

For convex ϕ: $\phi(\mathbb{E}[X]) \leq \mathbb{E}[\phi(X)]$ and also

$\forall \alpha_i \geq 0, \sum \alpha_i = 1$ and any $x_i > 0$: $\log(\sum_i \alpha_i x_i) \geq \sum_i \alpha_i \log(x_i)$

Matrix calculus (numerator layout)

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \vdots \\ \frac{\partial y_m}{\partial x} \end{bmatrix} \quad \frac{\partial x}{\partial y} = \begin{bmatrix} \frac{\partial x}{\partial y_1} & \cdots & \frac{\partial x}{\partial y_n} \end{bmatrix} \quad \frac{\partial y}{\partial z} = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \cdots & \frac{\partial y_m}{\partial z_n} \end{bmatrix}$$

$$\frac{\partial A}{\partial x} = \begin{bmatrix} \frac{\partial a_{11}}{\partial x} & \cdots & \frac{\partial a_{1n}}{\partial x} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_{m1}}{\partial x} & \cdots & \frac{\partial a_{mn}}{\partial x} \end{bmatrix} \quad \frac{\partial x}{\partial A} = \begin{bmatrix} \frac{\partial x}{\partial a_{11}} & \cdots & \frac{\partial x}{\partial a_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial x}{\partial a_{m1}} & \cdots & \frac{\partial x}{\partial a_{mn}} \end{bmatrix}$$

1D Derivation Rules

$\frac{\partial (f \cdot g)}{\partial x} = f' \cdot g + f g'$ $\quad \frac{\partial}{\partial x} \frac{f}{g} = \frac{f'g - g'f}{g^2}$ $\quad \frac{\partial f(g(x))}{\partial x} = f'(g(x))\, g'(x)$

$\frac{\partial x^r}{\partial x} = r \cdot x^{(r-1)}$ $\quad \frac{\partial}{\partial x} \frac{1}{f(x)} = \frac{-f'(x)}{(f(x))^2}$ $\quad (f^g)' = f^g \left(f' \frac{g}{f} + g' \ln f \right)$

Multivariate Chain Rule

For function $g: \mathbb{R}^n \to \mathbb{R}^m$, $x \mapsto y$ and $f: \mathbb{R}^m \to \mathbb{R}^k$, $y \mapsto z$ we have: $\frac{\partial z_k}{\partial x_j} = \sum_{i=1}^{m} \frac{\partial z_k}{\partial y_i} \frac{\partial y_i}{\partial x_j}$

$\frac{\partial z_k}{\partial x} = \frac{\partial z_k}{\partial y} \frac{\partial y}{\partial x}$ $\quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$

$\frac{\partial f(x(t), y(t))}{\partial t} = \frac{\partial f}{\partial x} \frac{\partial x}{\partial t} + \frac{\partial f}{\partial y} \frac{\partial y}{\partial t}$ $\quad$ x, y, t can be scalar, vector or matrix.

Derivatives:

$\frac{\partial}{\partial x}(b^Tx) = \frac{\partial}{\partial x}(x^Tb) = b$ $\quad \frac{\partial}{\partial x}(x^Tx) = 2x$ $\quad \frac{\partial}{\partial x}(x^TAx) = (A + A^T)x$

$\frac{\partial}{\partial x}(b^TAx) = A^Tb$ $\quad \frac{\partial}{\partial x}(c^TXb) = cb^T$ $\quad \frac{\partial}{\partial X}(c^TX^Tb) = bc^T$

$\frac{\partial}{\partial x}(\|x - b\|_2) = \frac{x - b}{\|x - b\|_2}$ $\quad \frac{\partial}{\partial x}(\|x\|_2^2) = 2x$ $\quad \frac{\partial}{\partial X}(\|X\|_F^2) = 2X$

$\frac{\partial}{\partial x} \log(x) = \frac{1}{x}$ $\quad \frac{\partial}{\partial x} \frac{1}{f(x)} = \frac{-f'(x)}{(f(x))^2}$ $\quad \frac{\partial}{\partial x}|x| = (-1)^{1_{(x<0)}}$

Kullback-Leibler Divergence

$D_{KL}(P \| Q) = -\sum_{x \in X} P(x) \log\left(\frac{Q(x)}{P(x)}\right) = \sum_{x \in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$

$D_{KL}(P \| Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$

Is not symmetric: $D_{KL}(p\|q) \neq D_{KL}(q\|p)$

Entropy: $H(P) = -\sum_i P_i \log P_i$

Cross entropy: $H(p, q) = \mathbb{E}_p(-\log(q)) = H(p) + D_{KL}(p\|q)$

Jensen-Shannon Divergence

$D_{JS}(P \| Q) = \frac{1}{2} D_{KL}(P \| \frac{1}{2}(P+Q)) + \frac{1}{2} D_{KL}(Q \| \frac{1}{2}(P+Q))$

NEURAL NETWORK BASICS

Perceptron learning: Init $w^{(0)} = 0$, then while $\exists y$ that is misclassified do $\{w^{k+1} = w^k + \eta(y - \hat{y})x^{i_k}$ where $\hat{y} = (\langle w^k, x^{i_k} \rangle > 0)\}$ If lin. sep. converges in finite time.

Sigmoid function: $\mathrm{sigm}(x) = 1/(1+e^{-x}) = e^x/(e^x+1)$

for $\sigma(w^Tx + b)$: 

Logistic regression (MLE): Given $D = \{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\}$

Model: $y^{(i)} \sim \mathrm{Bernoulli}(\sigma(w^Tx))$.

MLE: $w_{MLE} = \mathrm{argmax}_w P(D|w) = \mathrm{argmax}_w \prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, w)$

$= \mathrm{argmax}_w \prod_{i=1}^{n} [\sigma(w^Tx^{(i)})]^{y^{(i)}} \prod_{i=1}^{n} [1 - \sigma(w^Tx^{(i)})]^{1-y^{(i)}}$

(π_i abbreviates $\sigma(w^Tx^{(i)})$)

Neg. log-likelihood: $-\log P(D|w) = \sum_i y^{(i)} \log \pi_i + (1 - y^{(i)}) \log(1 - \pi_i)$

Softmax function: $\mathrm{softmax}(x_i) = e^{x_i} / \sum_i e^{x_i}$

Loss Functions

Squared Loss: $\frac{1}{2}(y - \hat{y})^2$

Cross entropy: $-y \log(\hat{y}) - (1-y) \log(1-\hat{y})$

Negative log-likelihood: $-\sum_k y_k \log \hat{y}_k$

Why not accuracy: is a step function, so a small change in weights might not change accuracy. Gradient is either zero or undefined.

MLE: Write down prob. dist, decompose into per sample prob., minimize negative log likelihood.

Universal Approximation Theorem

$\exists g(x)$ as NN (with non-linear activation), $g(x) \approx f(x)$ and $|g(x) - f(x)| < \varepsilon\ \forall x\ \forall f \in C^\infty(R)$ (continuous functions) (given enough hidden units, one layer is enough)

Training NN: Init weights to small random values, init biases to 0 or small positive values, update the params with GD. (non-lin. activation make loss non-convex)

TRAINING NEURAL NETWORKS

Regularization: Any technique which aims to reduce generalization error (not training error)

Data Standardization: $x_i' = (x - \mu)/\sigma$ where $\mu = \sum_{i=1}^{n} x_i$ and $\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu)^2}$. Make sure to use μ and σ for the test set!

Batch Normalization

Apply Batch Norm on each layer while computing $\mathcal{B}$'s μ over the entire batch.

$\mu = \frac{1}{n} \sum_{i=1}^{n} z_i$ $\quad \sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} (z_i - \mu)^2$ $\quad z_i^{norm} = \frac{z_i - \mu}{\sqrt{\varepsilon + \sigma^2}}$ $\quad \tilde{z}_i = \gamma z_i^{norm} + \beta$

with $\varepsilon > 0$ and γ, β learnable parameters

- Bias in linear / conv layer becomes redundant w/ BN
- Makes weights in deeper layers more robust to changes in shallow layers — slight regularization effect
- Test time: μ, σ estimated using exponential average of training time values

Data Augmentation: Generally more data leads to better performance, but acquiring training data is tedious (especially labels). Can generate synthetic data with a model-based approach, but that can lead to discrepancies between the real and synthetic data. Can also expand the data set by transforming existing samples to produce new samples (rotate, scale, noise injection (Gaussian blur, salt & pepper noise, channelwise dropout)), need to ensure consistency between transformed data sample and label. For classification introduces invariance $M(T(I)) = M(I)$, for regression introduces equivariance $M(T(I)) = T(M(I))$

Early Stopping: Save parameter where the lowest validation loss occurs then step if there is no improvement for p (patience) epochs. Then retrain model on entire training set for final test error (either for the same number of epochs (easiest & decent performance) or continue training).

As regularizer: Assuming $\|\nabla C(\theta)\| < K$ bounded, learning rate ε, initial parameters θ_0, optimal number of training iterations i^* and param. θ^* learned in i^* steps: $\|\theta_0 - \theta^*\|_2 < K\varepsilon i^*$.

Pseudocode early stopping

```
i, j, i* ← 0 ; v ← ∞ ; θ, θ* ← θ_0        Initialize
while j < p do:                            p: patience (how long to wait)
    θ ← train(θ, n) ; i ← i + n            train n steps
    v_c ← validate(θ)
    if v_c < v then
        j ← 0 ; θ* ← θ ; i* ← i ; v ← v_c
    else
        j ← j + 1
return θ*, i*                              optimal param. w.r.t. validation set
```

Dropout: Let $r_j^l \sim \mathrm{Bernoulli}(p)$ for $p \in [0,1]$ $\tilde{y}^l = r^l \circ y^l$, $z_i^{l+1} = w_i^{l+1} \tilde{y}^l + b_i^{l+1}$, $y_i^{l+1} = f(z_i^{l+1})$ (non-lin. act. func.)

Test time: Would need to evaluate $\sum_r p(r) p(y|x, \theta, r)$ (sum over exponentially many terms), but can instead rescale weights: $\tilde{W} = Wp$

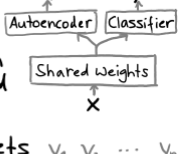
Bagging: Train many models on a bootstrapped dataset, then predict the test set on all models and aggregate (average, majority vote)

Differences to dropout: models are independent (not models sharing weights), train all models until convergence (not train only a small percentage of its models).

Batch Normalization: **Problem:** in SGD the distribution of x^l constantly changes (internal covariate shift), so we need to adapt constantly. Eliminating this speeds up training as no adaption is necessary.

Solution: standardize input distribution to each layer l: $\bar{x}_k^l = (x_k^l - \mathbb{E}[x_k^l]) / \sqrt{\mathrm{Var}[x_k^l]}$. As this loses some representational power, we perform a linear transform: $\tilde{x}_k^l = \gamma_k^l \bar{x}_k^l + \beta_k^l$ which makes the transform reversible (with learnable parameters γ and β). Note: some times $\sqrt{\mathrm{Var}[x_k^l] + \varepsilon}$ for small constant ε (division by 0)

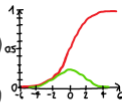
Semi-supervised Learning: Acquiring labeled data is expensive, but unlabeled data usually plentiful. Want to learn from both. Assumes $P(x)$ related to $P(y|x)$. Example of this:



Multitask learning: for datasets with similar samples and related labels. Divide model into shared and task-specific parameters. Assumes that $P(y_1|x)$ related to $P(y_2|x), \dots$

Activation Functions:

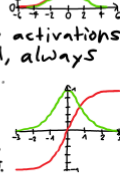
Sigmoid: $\sigma(x) = 1/(1+e^{-x})$, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$



+ interpretation as the firing rate of a neuron. Finite range → stable training
- saturates towards either end, so large activations get small gradients. Not zero-centered, always positive (cannot change "direction").

Tanh: $\tanh(x) = 2\sigma(2x) - 1 = (e^x - e^{-x})/(e^x + e^{-x})$

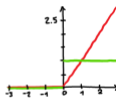
$\tanh'(x) = 1 - \tanh^2$



+ has finite range, zero-centered, has stronger steeper gradients than σ.
- activations also saturate.

Rectified Linear Unit: $\mathrm{ReLU}(x) = \max(0, x)$

$\mathrm{ReLU}'(x) = 1_{[x>0]} = 1$ if $x > 0$ else 0



+ Inexpensive operation, sparse activations, greatly accelerates convergence.
- can blow up the activation and destabalize training. "dying" ReLU problem: units with negative activations get no update

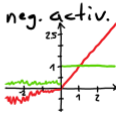
Leaky ReLU: with small pos. const. α:

$\mathrm{LReLU}(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ $\quad \mathrm{LReLU}'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

+ Attempts to solve "dying" ReLU by introducing small negative slope for neg. activ.

Randomized Leaky ReLU: $\alpha \sim U(a, b)$

$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ $\quad f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$



At test time set $\alpha = (a+b)/2$.

Optimization Algorithms:

Gradient Descent: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$. Follows the direction of the loss surface downhill.

SGD: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$. + prevents redundant calculations, enables to jump to new and potentially better local minima. − has risk of overshooting.

SGD Problem: Gradient big in the direction where we do not need to travel and small in the direction where we want to travel → oscillations, slow convergence

Mini-batch GD: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$
+ reduces the variance of the parameter updates which can lead to more stable convergence, can make use of highly optimized matrix operations.

Momentum: $V_t = \gamma V_{t-1} + \eta \nabla_\theta J(\theta)$, $\theta = \theta - V_t$. Instead of using the gradient to change the position of a "weight particle", use it to change the velocity. (helps with ravines i.e. areas where the surface curves more steeply in one dimension than the other dimension)

Nesterov accelerated gradient (NAG): $\theta = \theta - V_t$, $V_t = \gamma V_{t-1} + \eta \nabla_\theta J(\theta - \gamma V_{t-1})$. Looks ahead where the momentum would take it, then calculate there

AdaGrad
- Compute Gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^i;\theta), y^i)$
- Accumulate Sq. Grad: $r \leftarrow r + g \odot g$
- Compute Update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$
- Apply Update: $\theta \leftarrow \theta + \Delta\theta$

+ Greater progress in more gently sloped parts of θ-space
− Accumulation of r results in too early decrease in LR

RMS Prop
- Compute Gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^i;\theta), y^i)$
- Accumulate Sq. Grad: $r \leftarrow \rho r + (1-\rho) g \odot g$
- Compute Update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$
- Apply Update: $\theta \leftarrow \theta + \Delta\theta$

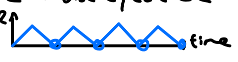
− ρ exponentially saturates at 1
+ Performs better in non-convex settings

Adaptive Moment Estimation (Adam):
$m_t = B_1 m_{t-1} + (1-B_1) g_t$ estimate of the first moment
$V_t = B_2 V_{t-1} + (1-B_2) g_t^2$ estimate of the second moment
$\hat{m}_t = m_t/(1-B_1^t)$, $\hat{V}_t = V_t/(1-B_2^t)$, $\theta_{t+1} = \theta_t - \eta / \sqrt{\hat{V}_t + \epsilon}\ \hat{m}_t$

Learning rate: If too large, can overshoot, if too low can be slow and get trapped in local minima.

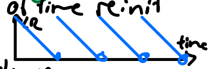
Step decay: decay learning rate by half after certain epochs.
Exponential decay: $\eta_{t+1} = \eta_0 e^{-kt}$
1/t decay: $\eta_{t+1} = \eta_0 / (1 + kt)$

Fast Geometric Ensembles
- Follow surfaces of constant loss to explore min
① 80% normal training ② 20% use cyclic LR



③ Save ckpt when lowest
④ Ensemble for inference

Stochastic Weight Averaging
① 80% normal training ② 20% of time reinit with w_{SWA} & adapt cyclic LR



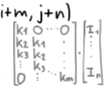
③ when LR lowest update w_{SWA}
$w_{SWA} \leftarrow \frac{w_{SWA} \cdot n_{models} + w}{n_{models} + 1}$
④ Use w_{SWA} for inference (Batch Norm has to be reinitialized for w_{SWA})

HMAX Model: simple (S) cells tuned to specific features ($y = \exp(-\frac{1}{2\sigma^2}\sum_{j=1}^{n_{s_k}}(w_j - x_j)^2)$), complex (C) cells combine output of various S cells to increase invariance and receptive field ($y = \max_{j=1,\dots,n_{c_k}} x_j$)

CONVOLUTIONAL NEURAL NETWORKS

Linear Transform: $T(\alpha u + \beta v) = \alpha T(u) + \beta T(v)$
Invariant to f: $T(f(u)) = T(u)$
Equivariant to f: $T(f(u)) = f(T(u))$
↳ any linear, shift-equivariant transform T can be written as a convolution.

Correlation: $I'(i,j) = \sum_{m=-k}^{k}\sum_{n=-k}^{k} K(m,n) I(i+m, j+n)$
Convolution: $I'(i,j) = \sum_{m=-k}^{k}\sum_{n=-k}^{k} K(-m,-n) I(i+m, j+n)$
Called shift-invariant if K does not depend on (i,j). Same for $K(i,j) = K(-i,-j)$ kernels. and kernels: matrix-vector multipl. $(I * K) =$



Differentiation: is a linear, shift invariant operation, can write as convolution: $\frac{\partial f}{\partial x} = \lim_{\varepsilon \to 0}\left(\frac{f(x+\varepsilon, y) - f(x,y)}{\varepsilon}\right) \approx \frac{f(x_{n+1}, y) - f(x_n, y)}{\Delta x}$ [-1 | 1]

CNN Backprop: $z^{(l)} = w^{(l)} * z^{(l-1)} + b^{(l)} = \sum_m \sum_n w_{mn}^{(l)} z_{i-m,j-n}^{(l-1)} + b^{(l)}$
Backward pass w.r.t. inputs

$\delta_{i,j}^{(l-1)} = \frac{\partial C}{\partial z_{i,j}^{(l-1)}} = \sum_{i'}\sum_{j'} \frac{\partial C}{\partial z_{i',j'}^{(l)}} \frac{\partial z_{i',j'}^{(l)}}{\partial z_{i,j}^{(l-1)}} =$ $(w_{00}^{(l)} z_{i,j-0}^{(l-1)} + \dots + w_{mn}^{(l)} z_{i-m,j-n}^{(l-1)} + \dots)$ ↳ everything vanishes except where $i' = i-m$, $j' = j-n$. What remains is w_{mn}, but as $i' = i-m \to m = i-i'$, $j' - j = n \to n = j - j'$

$= \sum_{i'}\sum_{j'} \delta_{i',j'}^{(l)} \frac{\partial}{\partial z_{i,j}^{(l-1)}}\left(\sum_m \sum_n w_{m,n}^{(l)} z_{i'-m,j'-n}^{(l-1)} + b\right)$

$= \sum_{i'}\sum_{j'} \delta_{i',j'}^{(l)} \cdot w_{i'-i,j'-j}^{(l)} = \delta^{(l)} * w_{-i,-j} = \delta^{(l)} * ROT_{180}(w^{(l)})$

Weight Update:
$\frac{\partial C}{\partial w_{m,n}^{(l)}} = \sum_i \sum_j \frac{\partial C}{\partial z_{ij}^{(l)}} \frac{\partial z_{ij}^{(l)}}{\partial w_{m,n}^{(l)}} = \sum_i\sum_j \delta_{ij}^{(l)} \frac{\partial}{\partial w_{mn}^{(l)}}\left(\sum_m\sum_n w_{mn}^{(l)} z_{i-m,j-n}^{(l-1)} + b^{(l)}\right)$
$= \sum_i \sum_j \delta_{ij}^{(l)} z_{i-m,j-n}^{(l-1)} = \delta^{(l)} * z_{-m,-n}^{(l-1)} = \delta^{(l)} * ROT_{180}(z^{(l-1)})$

CNN Backprop Exercise

Forward: X * K → Y = X*K → Y' = ReLU(Y) → Y'' = Pool(Y')

X:
```
0 0 1 1 0
0 1 0 1 1
1 1 1 1 0
1 0 1 1 1
1 1 0 1 1
```
K:
```
1 0 1
0 1 0
0 1 0
```
Y = X*K:
```
3 3 3
4 2 4
2 4 3
```
Y' = ReLU(Y):
```
3 3 3
4 2 4
2 4 3
```
Y'' = Pool(Y'): 4

↑ make sure to turn this 180°

Backward: ∂E/∂Y'' = 1 → ∂E/∂Y':
```
0 0 0
1 0 1
0 1 0
```
∂E/∂Y:
```
0 0 0
1 0 1
0 1 0
```
∂E/∂K:
```
3 1 3
2 3 2
2 3 1
```
∂E/∂X:
```
0 0 0 0 0
0 1 0 1 0
0 1 1 1 0
1 0 3 0 1
0 1 0 1 0
```

$\frac{\delta E}{\delta Y} \frac{\delta X * K}{\delta Y} = \frac{\delta E}{\delta Y} * rot_{180}(K) = \frac{\delta E}{\delta Y} \circledast K = rot_{180}(X \circledast \frac{\delta E}{\delta Y})$

$\frac{\delta E}{\delta Y} \frac{\delta X * K}{\delta X} = \dots = \frac{\delta E}{\delta Y} \circledast K$ with 2x zero padding

Pooling Layer: makes the representation smaller and more managable. $z^{(l)} = \max\{z_i^{(l-1)}\}$
$\frac{\partial z^{(l)}}{\partial z^{(l-1)}} = \begin{cases} 1 & \text{if } i = \text{argmax}_i\{z_i^{(l-1)}\} \\ 0 & \text{otherwise.} \end{cases}$ $\delta^{(l-1)} = \{\delta^{(l)}\}_{i^*}$ where $i^* = \text{argmax}_i\{z_i^{(l-1)}\}$

Deep Networks: many layers: large receptive field despite smaller filters (few parameters) (A deeper network should always perform at least as good as a shallow network. If not → failure to optimize) Can be improved with residual connections, reintroducing gradients.

1x1 Convolutions: reduces depth component which leads to fewer parameters, fewer computations. Also increases model capacity.

ResNet: Double convolution with ReLU form block → Stack many on top of each other

Semantic Segmentation: pixel-wise CV task. Using patches or all at once (fully convolutional) ↳ problem: convolutions at full res are expensive, so often uses down and then upsampling.

Unpooling: nearest neighbor (duplicate), bed of nails (only top left, rest 0), max unpooling (put where max was, else 0)

Transpose Convolution: (also de- & up-convolution, fractionally strided conv., backward strided conv.)
Output = copies of filter weighted by input, summed where it overlaps. Conv: $X * K = MX$ Conv^T: $X * K = M^T X$

Adversarial Defense: Adversarial training (brute force train with lots of adversarial samples to increase robustness), defensive distillation (train with "soft"-class probabilities instead of hard labels. Soft-probabilities stem from 2nd model trained on same task with hard labels. Smoothes network with respect to attack)

Convolutional Layers
kernel size $k_1 \times k_2$, padding $p_1 \times p_2$, stride $s_1 \times s_2$, dilation $d_1 \times d_2$
$H_{out} = (H_{in} + 2p_1 - d_1(k_1 - 1) - 1) s_1^{-1} + 1$
$W_{out} = (W_{in} + 2p_2 - d_2(k_2 - 1) - 1) s_2^{-1} + 1$

RECURRENT NEURAL NETWORKS

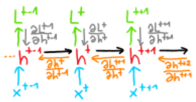
Dynamical system with inputs: $h^t = f(h^{t-1}, x^t; \theta)$
With f: transition function (same for all timesteps)
Instead of $g^t(x^{t-1}, \dots, x^1)$, has always same input size

RNN Types: one-to-one (vanilla RNN, character-level language model), one-to-many (image captioning), many-to-one (sentiment classification), many-to-many (delayed) (machine translation), many-to-many (immediate) (video classification on a frame level)

Vanilla RNN: $y^t = W_{hy} h^t$, $h^t = \tanh(W_{hh} h^{t-1} + W_{xh} x^t) = f(h^{t-1}, x^t, W)$
$L^t = \|\hat{y}^t - y\|^2$, $L = \sum_t L_t$.

Problems with long-term dependencies: (Intuition)
$h^{(t)} = W^T h^{(t-1)} = (W^t)^T h^{(t-1)}$ (note W to the power of t)
if ∃ eigendecomposition of W: $W = Q \Lambda Q^T$ then $h^{(t)} = ((Q\Lambda Q^T)^t)^T h^{(0)}$. If Q is orthogonal: $h^{(t)} = Q^T \Lambda^t Q h^{(0)}$
↳ if Λ gets small → vanishes, if Λ gets large → explodes

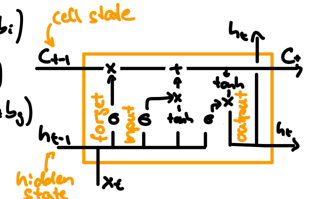
Backprop through time:
$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L^t}{\partial W}$, $\frac{\partial L^t}{\partial W} = \sum_{k=1}^{t} \frac{\partial L^t}{\partial y^t} \frac{\partial y^t}{\partial h^t} \frac{\partial h^t}{\partial h^k} \frac{\partial^+ h^k}{\partial W}$
Note: ∂^+ denotes the immediate derivative (keep h^{k-1} fixed)



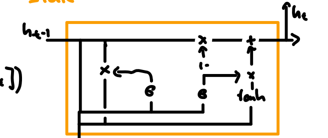
Vanishing/Exploding Gradient: We look at a small part of the gradient.
$\frac{\partial h^t}{\partial h^k} = \prod_{i=k+1}^{t} \frac{\partial h^i}{\partial h^{i-1}} = \prod_{i=k+1}^{t} W_{hh}^T \text{diag}[f'(h^{i-1})]$

Assume: λ_1 is the largest singular value of W_{hh} and $\|\text{diag}[f'(h^{i-1})]\| < \gamma$ where $\gamma \in \mathbb{R}$ (fine for tanh)
Case 1: It is sufficient $\lambda_1 < 1/\gamma$ for the vanishing gradient problem.
$\forall i, \left\|\frac{\partial h^i}{\partial h^{i-1}}\right\| \le \|W_{hh}^T\| \cdot \|\text{diag}[f'(h^{i-1})]\| < \frac{1}{\gamma}\gamma = 1$
Let $\eta \in \mathbb{R}$ s.t. $\forall i \left\|\frac{\partial h^i}{\partial h^{i-1}}\right\| \le \eta < 1$. We can see that $\left\|\prod_{i=k}^{t-k} \frac{\partial h^i}{\partial h^{i-1}}\right\| \le \eta^{(t-k)}$ ⟹ this goes to zero for $t \to \infty$.
Case 2: $\lambda_1 > 1/\gamma$ → exploding

LSTM
$i_t = \sigma(W_{xi} x_t + W_{hi} h_{t-1} + b_i)$
$f_t = \sigma(W_{xf} x_t + W_{hf} h_{t-1} + b_f)$
$o_t = \sigma(W_{xo} x_t + W_{ho} h_{t-1} + b_o)$
$g_t = \tanh(W_{xg} x_t + W_{hg} h_{t-1} + b_g)$
$C_t = f_t \odot C_{t-1} + i_t \odot g_t$
$h_t = o_t \odot \tanh(C_t)$



GRU
$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$
$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$
$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$
$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$



Gradient clipping: A way to avoid exploding gradients. Limit the length of the gradient vector

Image Captioning: Take some CNN (e.g. AlexNet), drop the fully connected part to label classes (not all FC), append a RNN. Get token probabilities, sample and feed in again (until reaches some END token)

GENERATIVE MODELS

Generative Modelling: Non-probabilistic (discriminative models) vs probabilistic (generative that are given random vector or discriminative that are also given input (conditionally generative))

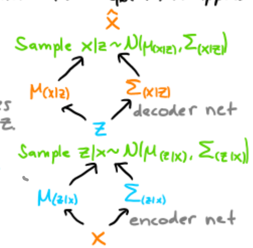
(Taxonomy of) Generative models
- Implicit density (highly expressive: universal, lack of theory & algorithms)
  - MarkovChain: GSN
  - Direct: GAN
- Explicit density
  - Tractable: BeliefNets, NADE/MADE, PixelRNN/CNN
  - Approximate
    - MarkovChain: Boltzmann Mach.
    - Variational: VAE

VARIATIONAL AUTOENCODER

Variational Autoencoders: probabilistic version that allows for sampling from the learned model to generate data with natural variation.
Assume training data D is generated from underlying latent distribution z. x: desired output (e.g. image)
Idea: sample from prior $p_{\theta^*}(z)$ z: latent factors that control then sample conditional $p_{\theta^*}(x|z^{(i)})$ generation process of x (e.g. scale)
↳ conditional complex (generate image) ⇒ param. with neural net.
↳ choose prior to be a simple distribution (e.g. Gaussian)

VAE Training: Want to capture this process via estimation of the parameters $p_\theta(x) = \int_z p_\theta(x|z) p(z) dz$
↳ problem: integral over continuous RV. Posterior also intractable
↳ solution: define additional encoder network $q_\phi(z|x)$ to approx. $p_\theta(x)$ using Bayes' rule.

$\mathcal{L}(x^{(i)}, \theta, \phi) = E_z[\log p_\theta(x^{(i)}|z)]$ (maximize reconstruction likelihood) → encourages samples of same category to be close in z.
$- D_{KL}(q_\phi(z|x^{(i)}) \| p_\theta(z^{(i)}))$ (make approximate posterior similar to prior)
↳ needed, otherwise model learns a zero variance (like autoencoder)
↳ encourage encoder to project latent space representations evenly around the center of the latent space



At run time, only use decoder network, sample z from prior

ELBO Derivation

start with data log-likelihood:

$$\log p_\theta(x^{(i)}) = \mathbb{E}_{z\sim q_\phi(z|x^{(i)})}[\log p_\theta(x^{(i)})] \quad (p_\theta(x^{(i)}) \text{ does not depend on } z)$$

$$\mathbb{E}_z\left[\log \frac{p_\theta(x^{(i)}|z)\cdot p_\theta(z)}{p_\theta(z|x^{(i)})} \cdot \frac{q_\phi(z|x^{(i)})}{q_\phi(z|x^{(i)})}\right] = \mathbb{E}_z[\log p_\theta(x^{(i)}|z)] - \mathbb{E}_z\left[\log \frac{q_\phi(z|x^{(i)})}{p_\theta(z)}\right] + \mathbb{E}_z\left[\log \frac{q_\phi(z|x^{(i)})}{p_\theta(z|x^{(i)})}\right]$$

$$= \mathbb{E}_z[\log p_\theta(x^{(i)}|z)] - D_{KL}(q_\phi(z|x^{(i)}) \| p_\theta(z)) + \underbrace{D_{KL}(q_\phi(z|x^{(i)}) \| p_\theta(z|x^{(i)}))}_{\geq 0}$$

Monte Carlo Gradient Estimator

Often encountered: $\nabla_\phi \mathbb{E}_{p_\phi(z)}[f(z)]$ with some differentiable function f. We also have a transformation $z = g(\varepsilon, \phi)$.

$$\nabla_\phi \mathbb{E}_{p_\phi(z)}[f(z)] = \nabla_\phi \int p_\phi(z) f(z)\, dz = \nabla_\phi \int p(\varepsilon) f(z)\, d\varepsilon$$

Note: this works because of the transformation and the differential area under the integral remains the same ($p_\phi(z)\,dz = p(\varepsilon)\,d\varepsilon$)

$$= \nabla_\phi \int p(\varepsilon) f(g(\varepsilon,\phi))\, d\varepsilon = \nabla_\phi \mathbb{E}_{p(\varepsilon)}[f(g(\varepsilon,\phi))] = \mathbb{E}_{p(\varepsilon)}[\nabla_\phi f(g(\varepsilon,\phi))]$$

We can then estimate this using: $\frac{1}{S}\sum_{s=1}^{S} \mathbb{E}_{p(\varepsilon)}[\nabla_\phi f(g(\varepsilon^{(s)},\phi))]$

Or in practice draw samples from the noise distribution and average the gradients.

D_{KL} Non-Negativity

$$-D_{KL} = -\int_x p(x) \log \frac{p(x)}{q(x)}\, dx = \int_x p(x) \log \frac{q(x)}{p(x)}\, dx \leq \log \int_x p(x) \frac{q(x)}{p(x)}\, dx = \log \int_x q(x)\, dx = \log(1) = 0$$

Trick: $\mathbb{E}(\varphi(x)) \leq \varphi(\mathbb{E}(x))$ iff φ is concave ($\log$ is)

Reparameterization Trick

(to enable VAE backprop)

Given Gaussian with μ and σ^2 $\quad z \sim N(\mu, \sigma^2)$

Assume $\exists$ underlying random variable $\epsilon \sim N(0,1)$

$z = \mu + \sigma\varepsilon = f(x, \varepsilon, \theta) \Rightarrow$ can take derivative w.r.t μ, σ (tells us how changes in μ, σ affects output for fixed ε)

Advantage of using N

when modelling $p_\theta(z)$ and $q_\phi(z|x^{(i)})$ with normal distributions with diagonal covariance matrix, it allows drawing easily from the prior & can calculate D_{KL} analytically

Equation for analytical D_{KL} with N:

For any $p(z) = N(\mu_p, \sigma_p^2 I)$ and $q(z) = N(\mu_q, \sigma_q^2 I)$

$$\int p(z)\log(q(z))\, dz = -\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\log\sigma_{qj}^2 - \frac{1}{2}\sum_{j=1}^{J}\frac{\sigma_{pj}^2 + (\mu_{pj} - \mu_{qj})^2}{\sigma_{qj}^2}$$

VAE Backprop

$$\mathbb{E}_z[\log p_\theta(x^{(i)}|z)] - D_{KL}(q_\phi(z|x^{(i)}) \| p_\theta(z))$$

$$= \mathbb{E}_z[\log p_\theta(x^{(i)}|z)] + \mathbb{E}_z\left[\log \frac{p_\theta(z)}{q_\phi(z|x^{(i)})}\right] = \mathbb{E}_z\left[\log \frac{p_\theta(z)\cdot p_\theta(x^{(i)}|z)}{q_\phi(z|x^{(i)})}\right] = \mathbb{E}_z\left[\log \frac{p_\theta(x^{(i)}, z)}{q_\phi(z|x^{(i)})}\right]$$

$$\nabla_{\theta,\phi}\mathbb{E}_{z\sim q_\phi}\left[\log \frac{p_\theta(x^{(i)},z)}{q_\phi(z|x^{(i)})}\right] = \nabla_{\theta,\phi}\mathbb{E}_{\epsilon\sim N(0,I)}\left[\log \frac{p_\theta(x, f(x,\epsilon,\phi))}{q_\phi(f(x,\epsilon,\phi)|x)}\right] = \mathbb{E}_\epsilon\left[\nabla_{\theta,\phi}\log \frac{p_\theta(x,f)}{q_\phi(f|x)}\right]$$

$$\approx \frac{1}{k}\sum_{i=1}^{k}\nabla_{\theta,\phi}\log(p_\theta(x,f)/q_\phi(f|x))$$

Limitations of VAEs

Tendency to generate blurry images (believed to be due to injected noise and weak inference models), recently better with inverse autoregressive flow

β-VAE

Learn disentangled representation without supervision, for $\beta=1$ same as VAE

$$\min: \mathcal{L}_\beta = -\mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] + \beta D_{KL}(q_\phi(z|x) \| p_\theta(z))$$

AUTOREGRESSIVE MODELS

Autoregression: for timeseries, $x_t = b_0 + b_1 x_{t-1} + b_2 x_{t-2}$ taking observations from previous timesteps as input. (e.g. sequence models)

Tabular approach: $p(x) = \prod_{i=1}^{n} p(x_i | x_1, \dots, x_{i-1})$ #params 2^{n-1}

can also specify $p(x_i|x_1,\dots,x_{i-1}) = \text{Ber}(f(x_1,\dots,x_{i-1}))$

Fully visible belief networks: $\hat{x}_i = p(x_i = 1 \mid x_1, \dots, x_{i-1})$ modeled via logistic regression: $f(x_1,\dots,x_{i-1}) = \sigma(\alpha_0^i + \dots + \alpha_{i-1}^i x_{i-1})$, has $O(n^2)$ parameters

Neural Autoregressive Density Estimator (NADE): autoencoder-like network to learn $p(x_i = 1 | x_1, \dots, x_{i-1})$

$h^{(i)} = \sigma(b + \underbrace{W_{:,<i}}_{\text{(first i cols)}} x_{<i})$, $\hat{x}_i = \sigma(c_i + \underbrace{V_{i,:}}_{\text{i-th row}} h^{(i)})$ for binary data

(We can use that $(b + W_{:,<i+1}x_{<i+1}) - (b + W_{:,<i}x_{<i}) = W_{:,i+1}x_{i+1}$).

Train by max. $\frac{1}{T}\sum_{t=1}^{T}\log(p(x^{(t)})) = \frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{D}\log p(x_i^{(t)} | x_{<i}^{(t)})$

Advantages: efficient (computations in $O(TD)$ could make use of second order optimizers, easily extendable to other types of observations (reals, multinomials). Random input order is fine.

Training: take data from training, **Inference:** sampling

Extensions: RNADE: conditionals modelled by mixture of Gaussians, DeepNADE: a DNN trained to assign a cond. distr. to any given var. given any subset of others (order-less), ConvNADE

Masked Autoencoder Distribution Estimator (MADE)

Constrain autoencoder s.t. output can be used as conditional $p(x_i|x_{<i})$. To fulfill autoregressive property: no computational path output unit x_d and any of the input units $x_1,\dots,x_D$ must exist. Can also be seen as masks $m_i^{(l)}$: 1 if unit i of layer l is connected to unit j of layer $l-1$. Train with NLL of binary x. Computing $p(x)$ is just forward pass, sampling requires D forward passes. In practice very large hidden layers necessary.

Generative Models for Natural Images

Use an explicit density model & use chain rule to decompose likelihood of image x into prod. of 1D distr. $p(x) = \prod_{i=1}^{n} p(x_i|x_{<i})$ → need to define ordering of pixels, complex distributions → paramet. via NN. Then max. LL.

PixelRNN

Generate pixel starting from the corner and dependency to prev. pixels modelled using LSTM

Issue: sequential gen s owl due to explicit pixel dependencies

PixelCNN

Still generate starting from corner, but model the dependency as CNN: (masked convolutions)

$p(x_i|x_{<i}) = p(x_{i,R}|x_{<i}) \cdot p(x_{i,G}|x_{i,R}, x_{<i}) \cdot p(x_{i,B}|x_{i,R}, x_{i,G}, x_{<i})$

stacks can be used to speed it up. Add information between vertical and horizontal stack (preserve pixel dependencies)

$h_{k+1} = \tanh(W_{k,f} * h_k) \odot \sigma(W_{k,g} * h_k)$. Training is faster than PixelRNN (can parallelize convolutions) but still slow (sequential generation).

Advantages: explicit likelihood, likelihood of training data is good eval. metric, good samples.

Dilated convolution

Receptive field size can increase exponentially with dilated layers.

WaveNet

Temporal convolutional networks uses dilated convolution.

↳cannot use strided conv, due to need to preserve resolution

Variational RNN (VRNN)

RNNs are autoregressive models Increase expressive power of RNNs by incorporating stochastic latent variables into hidden state of RNN Include one VAE per timestep. Including a dynamic prior explicitly models temporal dependencies between timesteps. Can include conditional (e.g. style and content of handwriting.

Stochastic Temporal Convolutional Networks (STCN)

Combines computational advantages of TCNs with expressiveness of stochastic RNNs: integrate a hierarchical VAE with a TCN.

Self Attention and Transformers

Transforms input Embeddings to Keys, Values, Queries

$K = XW_K \in \mathbb{R}^{T\times D}$ $\quad V = XW_V \in \mathbb{R}^{T\times D}$ $\quad Q = X_T W_Q \in \mathbb{R}^{1\times D}$

attention weights: $\alpha = \text{softmax}(QK^T/\sqrt{D} + M) \in \mathbb{R}^{1\times T}$

where $M = \begin{pmatrix} 0 & -\infty \\ 0 & 0 \end{pmatrix}$ masks out future influences

Prediction: $x_T = \alpha V$

NORMALIZING FLOWS

Have analytic model density $p_\theta(x)$ and latent variable space

$z \xrightarrow{f_\theta^{-1}} x$ $\quad x = f_\theta(z)$ $\quad z = f_\theta^{-1}(x)$

where f_θ is: deterministic & invertible, preserves dimensionality, Jacobian is efficient (block diagonal)

Change of variables

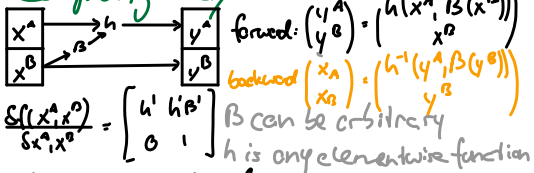
Maps one distribution to another

For a transformation $f(z) = x$

$$p_x(x) = p_z(f^{-1}(x)) \left|\det\left(\frac{\partial f^{-1}(x)}{\partial x}\right)\right|$$

$$= p_z(f^{-1}(x)) \left|\det\left(\frac{\partial f(x)}{\partial x}\right)\right|^{-1}$$

Coupling Layer



forward: $\begin{pmatrix} y^A \\ y^B \end{pmatrix} = \begin{pmatrix} h(x^A, \beta(x^B)) \\ x^B \end{pmatrix}$

backward: $\begin{pmatrix} x_A \\ x_B \end{pmatrix} = \begin{pmatrix} h^{-1}(y^A, \beta(y^B)) \\ y^B \end{pmatrix}$

β can be arbitrary

h is only elementwise function

$$\frac{\partial f(x^A, x^B)}{\partial x^A, x^B} = \begin{bmatrix} h' & h'\beta' \\ 0 & 1 \end{bmatrix}$$

We get complex transforms via composition

$x = f(z) = f_k \circ f_{k-1} \circ \dots \circ f_1$ Training: (exact log-likelihood)

$$\max \log p_x(x) = \sum_{x\in D}\left(\log p_z(f_1^{-1}(x)) + \sum_k \log\left|\det\left(\frac{\partial f_k^{-1}(x)}{\partial x}\right)\right|\right)$$

Inference: $p_x(x) = p_z(f_1^{-1}(x)) \prod_k \left|\det\left(\frac{\partial f_k^{-1}(x)}{\partial x}\right)\right|$

→ Also continuously via ODE: $\dot{x} = f_\theta(x(t), t)$

Multi-Scale Architecture

L Levels and K steps

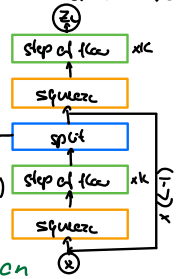
Squeeze: split input into channels by reducing the spatial resolution

flow step: ① Activation norm (like Batch/Norm: $y = sx + b$; $x = (y-b)/s$

② invertible 1x1 conv: h is a $H\times W\times C$ tensor, $W = PL(U + \text{diag}(s))$ (and rotation

$\log\left|\det\left[\frac{\partial}{\partial h}\text{conv}(h, W)\right]\right| = H\cdot W\cdot\log|\det(W)| = \sum(\log(s))$

③ (Conditional) Coupling Layer



Applications in Computer Vision

Super Resolution: Normalizing flow learns a dist of HR variants. Uses low resolution image as conditioning

Disentanglement: Conditions on learned attributes

C-Flow: Use two flows, condition one on the other. Each flow handles one representation, combining gives transfer

☹ Training is still slow

GENERATIVE ADVERSERIAL NETWORKS

GAN Intuition: Want to sample high dimensional data points from true data distribution (intractable), so draw from simple distribution and use a neural network to transform into output

Use generator $G: \mathbb{R}^D \to X$ and discriminator $D: X \to [0,1]$

GAN Objective: Consider only the discriminator

$\mathcal{L}(D) = -\frac{1}{N}\left(\sum_{i=1}^{N} y^{(i)} \cdot \log(D(x^{(i)})) + \sum_{i=1}^{N}(1-y^{(i)})\log(1 - D(x^{(i)}))\right)$

for GANs we know 50% are $y=1$ and 50% are $y=0$.

Our goal is to train D&G jointly:

$= \frac{1}{2}(\mathbb{E}_{x\sim p_d}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))])$

$= \frac{1}{2}(\mathbb{E}_{x\sim p_d}[\log D(x)] + \mathbb{E}_{x\sim p_g}[\log(1 - D(x))])$

$\Rightarrow$ a good G maximizes loss function of D.

Value function: $V(G,D) = \mathbb{E}_{x\sim p_d}[\log(D(x))] + \mathbb{E}_{x\sim p_g}[\log(1 - D(x))]$

$\Rightarrow G^*$ will fool any D: $G^* = \text{argmin}_G \max_D V(G,D)$

Optimum of $V(G,D)$ iff $p_d = p_g$: Let $D_G^* = \text{argmax}_D V(G,D)$ then the objective becomes $G^* = \text{argmin}_G V(G, D_G^*)$.

If $p_d = p_g$ then $D_G^* = \frac{p_d}{p_d + p_g}$. Proof: $V(G,D) = \int_x p_d(x)\log(D(x)) + \int_z p_z(z)\log(1 - D(G(z)))\, dz = \int_x p_d(x)\log(D(x)) + p_G(x)\log(1 - D(x))\, dx$

Optimal D: $\forall a,b \setminus \{0,0\}$ $f(y) = a\cdot\log(y) + b\cdot\log(1-y)$

$f'(x) = 0 \Leftrightarrow \frac{a}{y} - \frac{b}{1-y} = 0 \Rightarrow y = \frac{a}{a+b}$. $f''(\frac{a}{a+b}) = -\frac{a}{y^2} - \frac{b}{(1-y)^2} < 0 \Rightarrow D_G^* = \frac{p_d}{p_d + p_g}$

Obj. min. JS: $V(D_G^*, G) = \mathbb{E}_{x\sim p_d}[\log\frac{p_d}{p_d+p_g}] + \mathbb{E}_{x\sim p_g}[\log(1 - \frac{p_d}{p_d+p_g})]$

Global min is achieved if $p_d = p_g$ and at $\min V(\cdot,\cdot) = -\log 4$. Proof: $\mathbb{E}_{x\sim p_d}[\log\frac{p_d}{p_d+p_g} \cdot \frac{1}{2}] + \mathbb{E}_{x\sim p_g}[\log\frac{p_g}{p_d+p_g}\cdot\frac{1}{2}]$

$= -\log(2) + \mathbb{E}[\log(\frac{2\cdot p_d}{p_d+p_g})] - \log(2) + \mathbb{E}[\log(\frac{2\cdot p_g}{p_d+p_g})]$

$= -\log(4) + D_{KL}(p_d \| \frac{p_d+p_g}{2}) + D_{KL}(p_g \| \frac{p_d+p_g}{2})$ ↳ iff $p_d = p_g$

$= -\log(4) + 2\cdot D_{JS}(p_d \| p_g)$. Note: last term ≥ 0 then $= 0$

Jensen-Shannon Divergence

symmetric D_{KL}

$$D_{JS}(p\|q) = \frac{1}{2}D_{KL}\left(p \| \frac{p+q}{2}\right) + \frac{1}{2}D_{KL}\left(q \| \frac{p+q}{2}\right)$$

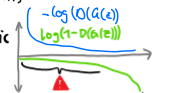
Obj: $\min_G \max_D \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$

GAN Training: $\min_G \max_D \mathbb{E}_{x\sim p(x)}[\log D_\theta(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D_\theta(G_\phi(z)))]$

Alternate: 1) $\max_D \mathbb{E}_{x\sim p(x)}[\log D_\theta(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D_\theta(G_\phi(z)))]$

(both GD) 2) $\min_G \mathbb{E}_{z\sim p_z(z)}[\log(1 - D_\theta(G_\phi(z)))]$

GAN Training Problems: In this problematic region G produces bad samples and gradient is very flat. Can lead to overtraining of the discriminator (G cannot catch up)



Solution: In 2. gradient ascent: $\max_\phi \mathbb{E}_{z\sim p(z)}[\log(D_\theta(G_\phi(z)))]$

Generally: need to find Nash-equilibrium in two-player game. This leads to: **mode collapse** (no sample diversity). happens if there is a saddle in dual energy landscape (the generator learns only variations of a single sample).

Issues with JS divergence: correlates badly with sample quality (do not know when to stop training), D tends to be optimal, JSD saturates → bad gradients

↳ solution: use Wasserstein distance

GAN Pseudocode

```
for #training iterations do
  for k steps do
    sample minibatch of m noise samples {z ~ p_g(z)}
    sample minibatch of examples {x} from data gen. dist.
    update discriminator by ascending on its stochastic grad:
      ∇_θd (1/m) Σ_{i=1}^m [log(D(x^(i))) + log(1 − D(G(z^(i))))]
    sample minibatch of m noise samples {z ~ p_g(z)}
    update the generator (ascent): ∇_θg (1/m) Σ_{i=1}^m log(D(G(z^(i))))
```

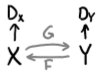
GAN: Comparison to VAE: No variational bound is needed, only samples. Generally sharper images. GANs are asymptotically consistent (not proven for VAE)

DC-GAN: G is an up-sampling network with deconvolutions, D is a CNN. Replace pooling by strided convolutions in D and strided deconvolutions in G. Use batch norm for both. Remove fully connected hidden layers, use ReLU in G except tanh for output, use LeakyReLU in D for all layers. Can do "vector math": smiling ♀ − neutral ♀ + neutral ♂ = smiling ♂ (image translation)

Pix2Pix: $\mathcal{L}(G,D) = \mathcal{L}_{cGAN}(G,D) + \lambda\mathcal{L}_{L1}(G)$ where $\mathcal{L}_{cGAN}(G,D) = \mathbb{E}_{x,y}[\log D(x,y)] + \mathbb{E}_{x,z}[1 - \log D(x, G(x,z))]$ and $\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x,z)\|_1]$

Limitations: requires paired images as training data ← unpaired image translation

Cycle GAN: $\mathcal{L}(G,F,D_X,D_Y) = \mathcal{L}_{GAN}(G,D_Y,X,Y) + \mathcal{L}_{GAN}(F,D_X,Y,X) + \lambda \mathcal{L}_{cyc}(G,F)$ where $\mathcal{L}_{cyc} = \mathbb{E}_{x\sim p_{data}(x)}[\|F(G(x))-x\|_1] + \mathbb{E}_{y\sim p_{data}(y)}[\|G(F(y))-y\|_1]$



Progressive Growing of GANs Grow G, D resolution by adding layers during training

Style GAN ① Learn an intermediate latent space ② Inject latent space (style) and noise in every layer of G

Diffusion Models Gradually add Gaussian Noise and learn to reverse it: $x_0 \rightleftarrows x_1 \rightleftarrows \dots \rightleftarrows z$

BODY MODELLING

Deep Features • Directly regress on (x,y) coordinates + refinement • Estimate Heatmap for each Joint Location with ConvNet

Thin-Slicing Networks ① predict Joint Heatmap ② Use flow warp to track joint location estimates over time ③ Minimize spring Energy in tracked body Graph over thin time slice

Linear Blend Skinning w_{ki}: Blend skinning weights
$t_i' = \sum_k w_{ki} G_k(\theta, J) t_i$ G_k: Rigid Bone transformation
J: Joint locations ⊖ Unrealistic Deformations

SMPL Body Model β: Body shape params
$t_i' = \sum_k w_{ki} G(\theta, J(\beta))(t_i + s_i(\beta) + p_i(\theta))$ (Offset of base mesh and scanned template)
$s_i(\beta)$: Shape correctives ⎫ Learned from
$p_i(\theta)$: pose correctives ⎭ template poses
⊕ Joints J(β) depend on shape β ⊕ pose/shape correctives

Optimization Based Fitting Directly optimize $\theta^* = \text{argmin} \|SMPL - 2D\ Joints\|^{\text{reproj}}$ ⊖ slow convergence
$\theta^{t+1}, \theta^t - \lambda\left(\frac{\delta L_{reproj}}{\delta\theta} + \frac{\delta L_{prior}}{\delta\theta}\right)$ ⊖ sensitive to init of θ, β

Learned Gradient Descent F_{NN}: Neural Net
$\theta^{t+1} = \theta^t + F_{NN}\left(\frac{\delta L_{reproj}}{\delta\theta}, \theta^t, x\right)$ $\frac{\delta L_{reproj}}{\delta\theta}$: Actual gradients
θ^t: current state x: Target pose; F_{NN} is learned by simulating 2D poses with Ground truth

Challenges ① Self-Occlusions ② Ambiguous Depth Information ③ Articulated Motion ④ Non-rigid Deformations

Template- & Regression Based Capture Template Mesh provides Data for challenging Poses.

NEURAL IMPLICIT SURFACES

Coordinate MLPs Takes fixed size positional Encoding as input, RGB output

3D Representations **Voxels** ⊖ Cubic in memory **Points** ⊖ Does not model connectivity **Meshes** ⊖ Difficult to work with

Implicit Differentiation Take Derivative on both sides of Equation, then substitute

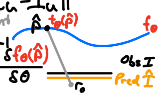
Neural Implicit Representations **Occupancy Networks** $f_\theta: \mathbb{R}^3 \times \mathcal{Z} \to [0,1]$ ← occupancy prob.
Signed Distance Fields $f_\theta: \mathbb{R}^3 \times \mathcal{Z} \to \mathbb{R}$ ← signed distance

Learning 3D Implicit Shapes From Watertight Meshes Randomly sample 3D points in space (inside / outside)
$\mathcal{L}(\theta,\psi) = \sum_{j=1}^{K} BCE(f_\theta(p_{ij}, z_i), o_{ij})$ ← Cross entropy Loss

From Pointclouds ⚠ Sparse supervision only on the surface. Eikonal PDE: $\|\nabla f(x)\| = 1;\ f(x) = 0\ \forall x \in \Omega$
→ solution gives distance from Ω
$\mathcal{L}(\theta) = \sum_{i\in I} |f_\theta(x_i)|^2 + \lambda \mathbb{E}_x(\|\nabla_x f_\theta(x)\| - 1)^2$
⊕ Converges to reproducing plane w/prob 1 for linear model

Differentiable rendering **Forward Pass** ① Find surface root $\hat{p}$ along ray w ② Evaluate texture field $t_\theta(\hat{p})$ **Secant Method**
$x_i = x_{i-1} - f(x_{i-1}) \frac{x_{i-1} - x_{i-2}}{f(x_{i-1}) - f(x_{i-2})}$ x_1, x_0 need to be on opposite sides of the root

Backward Pass $\mathcal{L}(\hat{I}, I) = \sum_u \|\hat{I}_u - I_u\|$
$\frac{\delta L}{\delta\theta} = \sum_u \frac{\delta L}{\delta \hat{I}_u} \cdot \frac{\delta \hat{I}_u}{\delta\theta}$ ↙ $\frac{\delta \hat{I}_u}{\delta\theta}$ depends on intersection point and texture value
$\frac{\delta \hat{I}_u}{\delta\theta} = \frac{\delta t_\theta(\hat{p})}{\delta\theta} + \frac{\delta t_\theta(\hat{p})}{\delta\hat{p}} \cdot \frac{\delta\hat{p}}{\delta\theta}$; $\frac{\delta\hat{p}}{\delta\theta} = -w\left(\frac{\delta f_\theta(\hat{p})}{\delta\hat{p}} \cdot w\right)^{-1} \frac{\delta f_\theta(\hat{p})}{\delta\theta}$



Neural Radiance Fields $F_\theta: (x,y,z,\theta,\phi) \to (R,G,B,\sigma)$
⊕ Can model transparency by predicting density
⊕ Conditioning on view (θ, φ) can model view dep. effects
↳ Only add view dir late to keep geometry consistent across views

Volume Rendering $\alpha_i = 1 - e^{-\sigma_i \delta_i}$ $\delta_i = t_{i+1} - t_i$ σ_i: network output
Transmittance: $T_i = \prod_{j=1}^{i-1}(1-\alpha_j)$ Color $c = \sum_{i=1}^{N} T_i \alpha_i c_i$

Training: $\mathcal{R}(\theta) = \sum_i \|render_i(F_\theta) - I_i\|^2$ ← RGB Loss
Faster with Hierarchical Sampling $c = \sum_{i=1}^{N} T_i \alpha_i c_i$
⊕ Can model transparency & thin structure
⊖ Generally worse geometry than implicit surfaces

Positional Encoding (Fourier Features) MLPs prefer smooth functions ⇒ Bad high frequency Details
Pass coordinates through random Fourier Features
$v \to \square\square\square \to y \Rightarrow$ (sin(v), cos(v), sin(2v), cos(2v), sin(4v), cos(4v), …) $\to \square\square\square \to y$ Can model a higher frequency function

CLASSICAL REINFORCEMENT LEARNING

Reinforcement Learning: Uses: games, robot control, attention based models, logistics & operations
Goal: achieve high cumulative reward over time.
Value function of policy π, V_π, expresses expected cumulative reward for each state when following π.
$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t=s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_t|S_t=s] = \sum_a \pi(a|s)\sum_{s'}\sum_r p(s',r|s,a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1}=s']]$
$= \sum_a \pi(a,s) \sum_{s'}\sum_r p(s',r|s,a)(r + \gamma V_\pi(s'))$ where $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$

MDP: states S, actions A, reward function r, transition function p, initial state s_0, discount factor γ

Value Iteration: Given a value function V_π, we can derive a greedy policy π': $\pi'(s) = \text{argmax}_{a\in A}(r(s,a) + \gamma V_\pi(p(s,a)))$. One can show $V_{\pi'}(s) \geq V_\pi(s)$. The optimal policy π* is greedy w.r.t. V_{π^*}. V_{π^*} fulfills the Bellman Optimality Equation: $V^*(s) = \max_{a\in A}\{r(s,a) + \gamma V^*(p(s,a))\}$. Can use step cost instead of γ. Update backwards each time doing the action that gives the highest reward until nothing changes anymore.

```
i←0, V0←0
while Δ>θ
  Δ←0
  for all s∈S
    v ← V(s)
    V(s) ← max_{a∈A} Σ_{s',r} p(s',r|s,a)[r+γV(s')]
    Δ ← max(Δ, |v−V(s)|)
πi ← greedy policy w.r.t. Vi
```

Theorem: value iteration converges. At convergence found the optimal V*. Need to find the optimal (stationary & greedy) policy.

Advantages of DP: Exact methods, policy/value iteration guaranteed to converge in finite number of iterations, value iteration typically more efficient easy implementation

Disadvantages of DP: need to know transition probability matrix, need to iterate over the whole state space (very expensive), requires memory proportional to size of state space, generally computationally expensive

Problem with large state space: we will never visit all states, but only a small subset. We cannot estimate the values of these states.

Monte Carlo Methods: value function is $\mathbb{E}[\cdot]$: $V_\pi(s) = \mathbb{E}_{a_t\sim\pi(s_t),\, s_{t+1}\sim p(s_t,a_t)} \sum_{t=0} \gamma^t r(s_{t+1}, a_{t+1})$, so we can just run policies with the given policy and compute samples of: $\sum_{t=0} \gamma^t r(s_{t+1}, a_{t+1})$
(follow policy, update values, update policy, repeat)
Advantages: Unbiased estimate, do not need to know system dynamics. **Disadvantages:** High variance, exploration/exploitation dilemma, need termination state (slow for long episodes)

Temporal Difference Learning: For each step with action a from s to s' that we take with our policy, we compute the difference to our current estimate and update our value function: $\Delta V(s) = r(s,a) + \gamma V(s') - V(s)$, $V(s) \leftarrow V(s) + \alpha\Delta V(s), \alpha > 0$
SARSA: On policy: Computes the Q-Value according to a policy and then the agent follows that policy.
Q-Learning: Off Policy: Computes the Q-Value according to a greedy policy, but the agent follows a different exploration policy:
$\Delta Q(S,A) = R_{t+1} + \gamma \max_a \{Q(S',a)\} - Q(S,A)$, $Q(S,A) \leftarrow Q(S,A) + \alpha\Delta Q(S,A)$
off policy since π for updating Q different from π' that we use for data collection.

Advantages: Less variance than Monte Carlo sampling due to bootstrapping. More sample efficient, do not need to know the transition probability matrix. **Disadvantages:** biased due to bootstrapping, exploration/exploitation dilemma, can behave poorly in stochastic environments. requires exploring full state-action space (does not work for continuous action space)

Taxonomy of RL methods
Policy Optimization → Grad-free methods (-Param. Perturbation, -Evolutionary strat.), Policy grad.
Methods based on TD learning → Policy iter., Value iter.
Policy grad. and Policy iter. → Actor-critic methods
Value iter. → Q-learning

DEEP REINFORCEMENT LEARNING

Policy gradients: $\pi(a_t|s_t) = \mathcal{N}(\mu_t, \sigma_t^2|s_t)$. We can collect rollout trajectories: $p(\tau) = p(s_1, a_1, \dots, s_T, a_T) = p(s_1)\prod_{t=1}^{T} \pi(a_t|s_t)\, p(s_{t+1}|a_t, s_t)$ then we update the policy: good traj. more & bad traj. less likely.

Policy gradient: goal: $\theta^* = \text{argmax}\, J(\theta)$ by updating $\theta \leftarrow \theta + \nabla_\theta J(\theta)$ with gradient ascent.
$J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}[\sum_t \gamma^t r(s_t, a_t)] = \mathbb{E}_{\tau\sim p_\theta(\tau)}[r(\tau)] = \int p(\tau) r(\tau)\, d\tau$
$\nabla_\theta J(\theta) = \int \nabla_\theta p(\tau) r(\tau)\, d\tau = \int \frac{p(\tau)}{p(\tau)} \nabla_\theta p(\tau) r(\tau)\, d\tau = \int p(\tau) \frac{\nabla p(\tau)}{p(\tau)} r(\tau)\, d\tau$
$= \int p(\tau) \nabla_\theta \log p(\tau) r(\tau)\, d\tau$ (since $\frac{d \log(u(x))}{dx} = \frac{u'(x)}{u}$) $= \mathbb{E}_{\tau\sim p(\tau)}[\nabla_\theta \log p(\tau) r(\tau)]$
$\log p(\tau) = \log p(s_1, a_1, \dots, s_T, a_T) = \log[p(s_1)\prod_{t=0}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|a_t, s_t)]$
$\Rightarrow \nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p(\tau)}[(\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t))(\sum_{t=0}^{T} \gamma^t r(s_t, a_t))]$
Valid even if reward is discontinuous or unknown sample space of path is a discrete set.
Gradient tries to increase probability with pos. r and decrease probability with small/neg. r.

Reinforce: sample trajectories τ_i by rolling out the policy, i.e. sampling a random action from the current policy until the end of the episode. Monte Carlo type of RL method that evaluates full trajectories/episodes to update policy
$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_i (\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i))(\sum_{t=0}^{T} \gamma^t r(s_t^i, a_t^i))$, $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
(unbiased estimate) **Variance problem:** this produces noisy gradients: If all cumulative rewards are positive, we make all trajectories more likely. If all traj. rewards are 0, we might decrease probability of good trajectories.
Baseline: To reduce variance, we can introduce a baseline $b(s_t^i)$ (produces okay trajectories, actually any function that does not depend on a_t, e.g. average reward or estimate of state-value function)
$r(\tau) = \sum_{t=0}^{T} \gamma^t r(s_t^i, a_t^i) - b(s_t^i)$. Still unbiased!

Actor-critic: Use bootstrapping for reward estimation:
$\nabla_\theta J(\theta) = \frac{1}{N}\sum_i \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)\,(r(s_t^i, a_t^i) + \gamma V_\theta(s_{t+1}^i) - V_\theta(s_t^i))$
Both π and V are represented by neural nets.