1 General Formula

Frobenius/L2 Norm $\|\theta\|_F = \sqrt{\sum_j \theta_j^2}$

L1 Norm $\sum_i |\theta_i|$

$\frac{d}{dx}(x^n) = nx^{n-1}$ $\quad$ $\frac{d}{dx}(f(x)g(x)) = f(x)g'(x) + f'(x)g(x)$

$\frac{d}{dx}(f(g(x))) = f'(g(x))g'(x)$ $\quad$ $\frac{d}{dx}\ln(x) = \frac{1}{x}$

$\frac{d}{dx}a^x = a^x\ln(a)$ $\quad$ $\frac{d}{dx}(\sin x) = \cos x$

$\frac{d}{dx}(\cos x) = -\sin x$ $\quad$ $\log(MN) = \log(M) + \log(N)$

$\log(M^k) = k\log(M)$ $\quad$ $\log_a b = \frac{\log_c b}{\log_c a}$

2 Neurons

Signals travel via axon, from Soma. Dendrites get signals.

Membranes have Na (out->in) and K Channels (in->out) and a Na-K Pump (3 Na in->out for 2K out->in).

Membrane Potential Difference in voltage across membrane. At Rest: –70mV, enforced by Na-K Pump.

Action Potential Electrical impulse travelling along axon to the synapse.

2.1 Hodgkin-Huxley Model

Model of Neuron based on Ion Channel components.

% of K+ channels open: $n(t)^4$. % of Na+ channels: $h(t)m(t)^3$

Gate dynamic system: $n(t) = \frac{dn}{dt} = \frac{1}{\tau_{n(V)}}\left(n_{\infty(V)} - n\right)$ (Same for all gates: n, h, m)

Membrane Potential Dynamics: $C\frac{dV}{dt} = J_{in} - g_{L(V-V_L)} - g_{Na}m^3h(V - V_{Na}) - g_K n^4(V - V_k)$

(C: Membrane capacitance) ($C\frac{dV}{dt}$: Net current inside cell) (J_{in}: Input current from other Neurons) (g_L: Leak conductance (membrane not impenetrable to ions)) (g_{Na}: Maximum Na conductance) (g_K: Maximum K conductance) (V_L: Resting potential)

2.2 Leaky-Integrate-and-Fire Model

Doesn't model spike shape, only considers sub-threshold voltage and threshold reached.

Dynamics system: $c\frac{dV}{dt} = J_{in} - g_L(V - V_L)$ Dimensionless equation is $\tau_m\frac{dv}{dt} = v_{in} - v$ for $v < 1$

Spike when $v = 1$ with t_{ref} wait before starting at 0 again.

Explicit Model: $v(t) = v_{in}\left(1 - e^{-\frac{t}{\tau}}\right)$ if v_{in} is constant. $V_{in} = RJ_{in}, \tau_m = RC, v = \frac{V - V_L}{V_{th} - V_L}, v_{in} = \frac{V_{in}}{V_{th} - V_L}$

Firing Rate: $\frac{1}{\tau_{ref} - \tau_m \ln(1 - 1/v_{in})}$ for $v_{in} > 1$

Tuning Curve: Graph showing how neuron reacts to different input currents.

2.3 Activation Functions (Sigmoidal)

Logistic Curve: $\frac{1}{1+e^{-x}}$ Range(0,1) | $\arctan(h)$ Range$(-\frac{\pi}{2}, \frac{\pi}{2})$ | $\tanh(z)$ Range$(-1, 1)$ | Threshold | ReLU | Softplus: $\log(1 + e^x)$

Smooth approx of ReLU | SoftMax: $\frac{\exp(z_j)}{\sum_j \exp(z_j)}$ | ArgMax - Largest to 1, 0 else

3 Synapses

Presynaptic action potential releases neurotransmitters across synaptic cleft which binds to postsynaptic receptors, opens ion channels.

Postsynaptic potential filter/Current entering **postsynaptic neuron**: $h(t) = kt^n e^{-\frac{t}{\tau_s}}$ if $t \geq 0 (n \in \mathbb{Z}_{\geq 0})$ 0 otherwise

k is selected so $\int_0^\infty h(t)dt = 1 \Rightarrow k = \frac{1}{n!\tau_s^{n+1}}$

Spike train: Series of multiple spikes $a(t) = \sum_{p=1}^k \delta(t - t_p)$

Dirac function: Infinite at $t = 0$, 0 everywhere else. Properties: $\int_{-\infty}^{\infty}\delta(t)dt = 1$, $\int_{-\infty}^{\infty} f(t)\delta(t-\tau)dt = f(\tau)$

Filtered Spike Train: $s(t) = a(t) * h(t)$ Essentially, convolve each spike in spike train to the postsynaptic potential filter.

Derivation:

$$s(t) = a(t) * h(t) = \int \sum_p \delta(t - t_p)h(t - \tau)d\tau = \sum h_{p(t-t_p)}$$

Neuron Activities Neurons have multiple connections with different weights represented by $W \in R^{N\times M}$. Neuron function: $\vec{z} = \vec{x}W + \vec{b}$, and $\vec{y} = \sigma(\vec{z})$

Bias Add neuron with constant 1, with weights: $\vec{x}W + \vec{b} = \begin{pmatrix}\vec{x} & 1\end{pmatrix} \cdot \begin{pmatrix}W \\ \vec{b}\end{pmatrix}$

Connections between spiking neurons Let $n = 0$ for simplicity in $h(t)$, this solves IVP $\tau_s\frac{ds}{dt} = -s, s(0) = \frac{1}{\tau_s}$

3.1 Full LIF Model

Dynamics are described by:

$$\begin{cases}\tau_m\frac{dv_i}{dt} = s_i - v_i \text{ if not in refractory period} \\ \tau_s\frac{ds_i}{dt} = -s_i\end{cases}$$

If $v_i = 1$, start refractory period, send spike, reset v_i to 0. If spike arrives from neuron j, then $s_i \leftarrow s_i + \frac{w_{ij}}{\tau_s}$

4 Universal Approximation Theorem

$\forall f \in C(I_n), \wedge \epsilon > 0, \exists G(x) = \sum_{j=1}^N a_j\sigma(w_j x + \theta_j)$ such that $|G(x) - f(x)| < \epsilon \forall x \in I_n$, $I_n \in [0, 1]$. $\sigma(wx)$ is any sigmoid function, with infinite weight becomes threshold. Piece functions are differences of thresholds.

5 Loss functions

Single Error: $L(y, t)$ is error between one output y and target t

Dataset Error: $\mathbb{E} = \frac{1}{N}\sum_{i=1}^N L(y_i, t_i)$ as average error over entire dataset.

Mean Squared Error: *Single Error*: $L(y, t) = \frac{1}{2}\|y - t\|_2^2$ *Activation Function*: ReLU *Problems*: Linear regression.

Bernoulli Cross Entropy: *Single Error*: $L(y, t) = -\log(P(y, t)) = -\log(t^y(1-y)^{1-t}) = -(t\log(y) + (1-t)\log(1-y))$ *Activation Function*: Logistic *Problems*: Output in range $[0, 1]$;

Categorical Cross Entropy: *Single Error*: $L(y, t) = -\log(P(x \in C_k, t)) = -\log\left(\prod_{k=1}^K (y^k)^{t^k}\right) = -\sum_{k=1}^k t^k\log(y^k)$ *Activation Function*: Softmax *Problems*: Classification with one-hot vectors

6 Gradient Descent

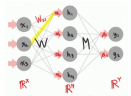
Goal: Minimize $E(\theta)$. Let $\nabla_\theta E = \begin{pmatrix}\frac{\partial E}{\partial\theta_1} & \cdots & \frac{\partial E}{\partial\theta_n}\end{pmatrix}$. Gradient Descent: $\theta_{n+1} = \theta_n - k\nabla E(\theta_n)$

7 Error Backpropagation

$\nabla_{z^{(l+1)}}E = \frac{\partial E}{\partial z^{(l+1)}}$

$h^{(l+1)} = \sigma\left(z^{(l+1)}\right) = \sigma\left(W^{(l)}h^{(l)} + b^{(l+1)}\right)$

$\nabla_{z^{(l)}}E = \frac{dh^{(l)}}{dz^{(l)}} \odot \left[\nabla_{z^{(l+1)}E\cdot\left(W^{(l)^T}\right)}\right]$ We transpose because W_{ij} is connection from ith node in l to jth node in $l+1$



Note that $\vec{a} = \vec{x}W$ in this diagram:

$\frac{\partial E}{\partial W_{ij}^{(l)}} = \frac{\partial E}{\partial z_j^{(l+1)}}\cdot\frac{\partial z_j^{(l+1)}}{\partial W_{ij}^{(l)}} = \frac{\partial E}{\partial z_j^{(l+1)}}\cdot h_i^{(l)}$

Finally, $\nabla_{z^{(l)}}E = \sigma'\left(z^{(l)}\right) \odot \left(\nabla_{z^{(l+1)}}E \cdot \left(W^{(l)}\right)^T\right)\nabla_{W^{(l)}}E = \left[h^{(l)}\right]^T\nabla_{z^{(l+1)}}E$

Vectorization: Generalize this process for a batch of samples by letting x be a matrix of samples instead of just one sample.

8 Auto-Differentiation

Each operation is a square with its variable dependencies. Each variable has a pointer to creator(operation that created it)

Pseudocode: Function $f = g(x, y, ...)$: 1) Create g Op object. 2) Save references to args $x, y, ...$ 3) Create Var for output f. 4) Set g.val as $g(x, y, ...)$. 5) Set f.creator to the g Op.

8.1 Differentiate

With each object in our graph, store derivative of total expression with respect to itself in member *grad* Use chain rule with parent operation Op.grad to get current grad. Ex: If y is parent of x, then x.grad = y.grad $\cdot \frac{\partial y}{\partial x}$ Add when multiple branches converge.

Backward method:

```
Var.backwards: self.grad += s; self.creator.backward(s)

Op.backward(s): for x in self.args:
x.backward(s * partialDeriv(Op, x))
```

In Var, self.val, self.grad, s must have same shape. In Op, s must be shape of operation output

9 Neural Nets w/w Auto-Diff

9.1 Neural Learning

Optimizing our weights and biases for our loss function. $W \leftarrow W - \kappa\nabla_W E$ By making network with AD classes, we leverage backward() to optimize gradient computation.

9.1.1 Pseudocode

Given Dataset (X, T) and network model **net**, with parameters θ and loss function L

- y = **net**(X)
- loss = L(y, T)
- loss.zero_grad()
- loss.backward()
- $\theta \leftarrow \kappa \cdot \theta$.grad

10 Overfitting

Overfitting: Big discrepancy in accuracy between training and testing or wayyyy too small training error.

Validation: Subset of training set for optimizing hyperparameters. **Weight Decay**: Loss considers hyperparameters. $\tilde{E}(\hat{y}, t; \theta) = E(\hat{y}, t; \theta) + \frac{\lambda}{2}\|\theta\|_F^2$ Updates are $\nabla_{\theta_i}\tilde{E} = \nabla_{\theta_i}E + \lambda\theta_i$ and $\theta_i \leftarrow \theta_i - \kappa\nabla_{\theta_i}E - \kappa\lambda\theta_i$

Data Augmentation: New samples by slight changes. **Dropout**

11 Optimization (Better learning rate)

Stochastic Gradient Descent Computing gradient of loss can be expensive for huge dataset. $E(Y, T) = \frac{1}{D}\sum_D L(y_i, t_i)$

Solution: Take random group γ of B samples. Estimate $E(Y, T) \approx E(\tilde{Y}, \tilde{T}) = \frac{1}{B}\sum_B L\left(y_{\gamma_b}, t_{\gamma_b}\right)$

Update after each batch B, and each epoch is gathering batches until entire dataset has been sampled.

11.1 Momentum

Use: Gradient descent oscillates often reaching optimum; grad.desc stops at local optimum. Gradient is a force instead of slope. $\theta_{n+1} = \theta_n + \Delta t v_n$ and $v_{n+1} = (1 - r)v_n + \Delta t A_n$ where A_n represents the gradient vector, so we make $v^{(t)} \leftarrow \beta v^{(t-1)} + \nabla_W E$ and $w^{(t)} \leftarrow w^{(t-1)} - \eta v^{(t)}$

v_n is "velocity" and A_n (gradient vector) is force, with direction and magnitude. By move in same direction, we gather "momentum", and increase velocity, taking bigger steps.

12 Deep Neural Networks

Vanishing Gradients When weights and biases are too high, the input currents become too high, and then derivatives are too small, so when you chain them across multiple layers, the gradients reduce severely. $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ with max 0.25, so when chained, each layer shrinks by factor of 4. So the closer the layer to the input, the smaller the gradient.

Exploding Gradients If weights are large and biases position the inputs in the high-slope region of the logistic function, then each layer can amplify the gradient, causing exploding gradients.

13 Convolutional Neural Networks

Convolution Operation that combines two functions by sliding them against each other.

Continuous Convolution $(f * g)(x) = \int_{-\infty}^{\infty} f(s)\cdot g(x - s)ds$

Discrete Convolution $(f * g)_m = \sum_{n=0}^{N-1} f_n \cdot g_{m-n}$

Convolution in 2D: $(f * g)_{m,n} = \sum_{ij} f_{ij}\cdot g_{m-i,n-j}$ $(f \circledast g)_{m,n} = \sum_{ij} f_{ij}\cdot g_{i-m,j-n}$

Each kernel is convolved against the layer before it, creating an activation map, creating a tensor for the next layer. We can have multiple kernels per input layer.

Note: The bias is attributed to the kernel as a hole, rather than the neurons in the kernel. 1x1 convolution layers are useful in multi-layered input. (Ex 1x1x64)

14 Hopfield Networks

Hopfield Network Given a network of N neurons (each connected to all others) a set of M possible targets, we want the network to converge to the nearest target.

$x_i = -1$ if $\vec{x}W + b_i < 0$, otherwise 1

Goal: Min Hopfield Energy w/w grad.desc, instead of backprop (cycles).

$E = -\frac{1}{2}\sum_i\sum_{j\neq i} x_i W_{ij}x_j - \sum_i b_i x_i = -\frac{1}{2}\vec{x}W\vec{x}^T - \vec{b}\vec{x}^T$, $W_{ii} = 0$

$\frac{\partial E}{\partial x_j} = -\sum_{i\neq j} x_i W_{ij} - b_j$

$\nabla_{\vec{x}}E = -\vec{x}W - \vec{b} \Rightarrow \tau_x\frac{d\vec{x}}{dt} = \vec{x}W + \vec{b}$

If $i \neq j$, $\frac{\partial E}{\partial W_{ij}} = -x_i x_j$ and if $i = j$ $\frac{\partial E}{\partial W_{ii}} = -x_i^2 = -1$

Gradient vector is $\nabla_W E = -\vec{x}^T\vec{x} + I_{N\times N}$

$\nabla_W E = -\frac{1}{M}\sum_{s=1}^M \left(\vec{x}^{(s)}\right)^T\vec{x}^{(s)} + I = -\frac{1}{M}X^T X + I$

Thus $\tau_w\frac{dW}{dt} = \frac{1}{M}X^T X - I$ $X^T X$ is coactivation states b/w neuron pairs.

15 Recurrent Neural Networks

Hidden layer encodes input sequence, for modeling sequential data.

Backprop Through Time Unroll network through time, to create feedforward network into a DAG, and evaluate the targets in sequence similar to how you would batch inputs.

Note: $h^i = f(x^i U + h^{i-1}W + b)$, $y^i = \text{Softmax}(h^i V + c)$

U: Input->Hidden. W: Hidden->Hidden V Hidden->Output Error function is: $E(y_1, ..., y_t, t_1, ..., t_t) = \sum_{i=1}^T L(y_i, t_t)\alpha_i$

The goal is to minimize the following:

$\min_\theta \mathbb{E}[E(y_1, ..., y_t, t_1, ..., t_t)]_{y,t}$

Objective: $\theta^* = \arg\min_\theta \mathbb{E}_{(x,t)\in D}\mathcal{L}(y, t)$, $\theta = \{U, V, W, b, c\}$

Deep RNN: Multiple Hidden layers per timestep. Each layer processes, then passes to next. Deep RNNS reduce cost compared to increasing hidden state size, and has better representation learning.

16 Gated Recurrent Units

RNNs can't capture long-range dependencies, because of vanishing gradients, so early tokens have small impact on later. Consider $h^n = wh^{n-1} + x^n$. If $|w| < 1$ then w^n shrinks exponentially as n grows, early info has less weight on h^n over time. If $|w| > 1$, h^n grows exponentially $\rightarrow$ unstable training.

16.1 Gated Recurrent Units

Gate mechanisms control which information to retain.

Candidate Hidden State: $\widetilde{h^n} = \tanh\left(\overrightarrow{h^{n-1}}W + \overrightarrow{x^n}U + \vec{b}\right)$. W : hidden->hidden, U : input->hidden, b : bias, $\tanh$: ensures output $\in (-1, 1)$

Gate Mechanism: $\overrightarrow{g^n} = \sigma\left(\overrightarrow{h^{n-1}}W_g + \overrightarrow{x^n}U_g + \vec{b_g}\right)$. σ means $g_i^n \in (0, 1)$

Final Hidden State Update: $\overrightarrow{h^n} = \overrightarrow{g^n} \odot \widetilde{h^n} + (1 - \overrightarrow{g^n}) \odot \overrightarrow{h^{n-1}}$

$\overrightarrow{g^n}$ controls how much of new candidate is retained, and $(1 - \overrightarrow{g^n})$ shows how much of previous state is preserved.

When $g = 0$ we don't update hidden state (meaning the word added no new context), if $g = 1$, then we do update, proving it's important information.

16.2 Full GRU

Update + reset gate added same as gate mechanism above. Reset tries to "forget" past hidden state for new candidate. When $\vec{r^n} \approx 0$, past is discarded.

Hidden state update: $\widetilde{h^n} = \tanh\left(\left(\overrightarrow{h^{n-1}} \odot \vec{r^n}\right)W + \overrightarrow{x^n}U + \vec{b}\right)$

$\overrightarrow{h^n} = \overrightarrow{g^n} \odot \widetilde{h^n} + (1 - \overrightarrow{g^n}) \odot \overrightarrow{h^{n-1}}$

16.3 minGRU

For N-length sequence, each component computed sequentially with poor parallelization w/w GPUs. $\overrightarrow{g^n} = \sigma\left(\overrightarrow{x^n}U_g + \vec{b_g}\right)$

$\widetilde{h^n} = \overrightarrow{x^n}U + \vec{b}$

$\overrightarrow{h^n} = \overrightarrow{g^n} \odot \widetilde{h^n} + (1 - \overrightarrow{g^n}) \odot \overrightarrow{h^{n-1}}$

Parallel Scan: $\overrightarrow{h^n} = A^n \odot \overrightarrow{h^{n-1}} + B^n$. $\in O(\log N)$

17 Autoencoders

Neural network that learns to encode/decode a set of inputs automatically into a smaller latent space.

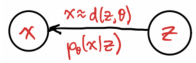
The model is trained using loss function minimizing $L(x', x)$ where x' is reconstructed input and x is original input.

To simplify latent space encoding, we can tie the weights of the encoder and decoder, so that the encoder weights are W and the decoder weights are W^T

18 Variational Autoencoders

Goal: Autoencoder that generates reasonable samples not in training set by encoding latent probability distribution.

Input: x. Latent space: z. Encoding: $p(z)$. Decoding: $d(z, \theta)$. Want $\max_\theta p(x) = \int p_{\theta(x\mid z)}p(z)dz$ (decoding probability)



Assume $p(x \mid z)$ is Gaussian, then NLL is $-\ln p_{\theta(x\mid z)} = \frac{1}{2\hat{\sigma}^2}\|X - d(z, \theta)\|^2 + C$ to maximize. Our goal is $\max_\theta \mathbb{E}_{z\sim p(z)}[p_\theta(x \mid z)]$ Since we don't know how to sample $p(z)$, we assume distribution $q(z)$ to approximate value.

$p(x) = \mathbb{E}_{z\sim p}[p(x \mid z)] = \sum_{z\sim p} p(x \mid z)p(z)$

$= \sum_{z\sim q} p(x \mid z)\frac{p(z)}{q(z)}q(z) = \mathbb{E}_{z\sim q}\left[p(x \mid z)\left(\frac{p(z)}{q(z)}\right)\right]$

Then NLL is: $-\ln p(x) \leq -\mathbb{E}_{q(z)}[\ln p(x \mid z)) + \ln\frac{p(z)}{q(z)}]$, by Jensen's inequality (If f is convex, $Ef(x) \geq f(Ex)$).

Then we simplify to $-\ln p(x) \leq KL(q(z) \parallel p(z)) - \mathbb{E}_{q(z)}[\ln p(x \mid z)]$.

$KL(q(z)\|p(z)) = -\mathbb{E}_{z\sim q}\left[q(z)\ln\left(\frac{p(z)}{q(z)}\right)\right]$.

Then choose $p(z) \sim \mathcal{N}(0, I)$ and we want to $\min_q KL(q(z))\|\mathcal{N}(0, I))$. Our encoder will be designed to have outputs μ, σ (not actual mean/std.dev) just parameters for distribution.

Note: $KL(\mathcal{N}(\mu, \sigma))\|\mathcal{N}(0, I)) = \frac{1}{2}(\sigma^2 + \mu^2 - \ln(\sigma^2) - 1)$

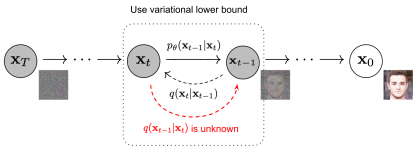
Note: $\mathbb{E}_q[\ln p(x \mid z)]$ can be written as $\mathbb{E}_q[\ln p(x \mid \hat{x})]$ where $x = d(z, \theta)$ and $z = \mu(x, \theta) + \varepsilon \odot \sigma(x, \theta)$. This reparameterszation trick was done so we can backprop on x, θ, by making the ε a separate noise vector.

Procedure:

1. Encode x by finding $\mu(x, \theta)$ and $\sigma(x, \theta)$
2. Sample $z = \mu + \varepsilon\sigma, \varepsilon = \mathcal{N}(0, I)$
3. Calc KL Loss $\frac{1}{2}(\sigma^2 + \mu^2 - \ln\sigma^2 - 1)$
4. Decode $\hat{x} = f(x, \theta) = d(z)$ using decoder
5. Calc loss $(L(x, \hat{x}) = \frac{1}{2}\|\hat{x} - x\|^2$ for Gaussian; $\sum_x x\ln\hat{x}$ for Bernoulli)
6. Gradient descent on $E = \mathbb{E}_x[L(x, \hat{x}) + \beta(\sigma^2 + \mu^2 - \ln\sigma^2 - 1)]$

19 Diffusion Models

Gen. model using noise-seeds to make new images, by training on model that noises samples so it learns to reverse noise.



Forward process: $q(\vec{x_t} \mid \overrightarrow{x_{t-1}}) = \mathcal{N}\left(\vec{x_t}; \sqrt{1-\beta_t}\overrightarrow{x_{t-1}}, \beta_t I\right)$. Final state is pure noise ($x_T \sim \mathcal{N}(0, I)$). The variance schedule $\beta_1, ..., \beta_T$ are hyperparameters controlling noise addition. Usually starts small, gets larger.

Unwrapping x_t: $\vec{x_t} = \sqrt{1-\beta_t}\overrightarrow{x_{t-1}} + \sqrt{\beta_t}\varepsilon_t \Rightarrow x_t = \sqrt{\alpha_t}x_0 + \sum_i c_i\varepsilon_i$ where $a_t \equiv 1-\beta_t$, $\overline{\alpha_t} \equiv \alpha_1 ... \alpha_t$ and $\varepsilon_i \sim \mathcal{N}(0, I)$. Since sum of gaussians is gaussian, $x_t = \sqrt{\overline{\alpha_t}}x_0 + \sqrt{1-\overline{\alpha_t}}\varepsilon$.

To get x_0 from x_t, we use neural net $\varepsilon_\theta(x_t, t)$ to estimate noise ε

To do reverse diffusion (generate input from noise), we need to minimize KL-divergence loss of $D_{KL}\left(q(x_{t-1}|x_t, x_0) \parallel p_{\theta(x_{t-1}|x_t)}\right)$ for $t = 2, ..., T$.
$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}|\mu_q, \Sigma_q), \mu_q = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\varepsilon_t\right)$

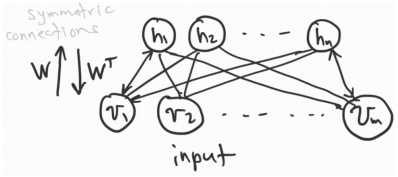
We assume parameterized probability density p_θ is Gaussian, so: $p(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}|\mu_\theta, \Sigma_\theta), \mu_\theta = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\varepsilon_{\theta(x_t,t)}\right)$
Then, KL-Divergence simplifies to $\mathbb{E}_{x_0,\varepsilon}\left[\lambda_t \left\|\varepsilon - \varepsilon_{\theta(x_t,t)}\right\|_2^2\right]$ where $\lambda_t = \frac{\beta_t^2}{2\sigma_t^2\alpha_{t(1-\overline{\alpha_T})}}$.
By substituting x_t we get $\mathbb{E}_{x_0,\varepsilon}\left[\left\|\varepsilon - \varepsilon_{\theta\left(\sqrt{\overline{\alpha_t}}x_0 + \sqrt{1-\overline{\alpha_t}}\varepsilon\right)}\right\|_2^2\right]$

Training Algorithm: Repeat until converge:
1. Take $x_0 \in D$, $t \sim \text{Uniform}(\{1, ..., T\})$, $\varepsilon \sim \mathcal{N}(0, I)$
2. Do grad.desc on $L = \left\|\varepsilon - \varepsilon_{\theta\left(\sqrt{\overline{\alpha_t}}x_0 + \sqrt{1-\overline{\alpha_t}}\varepsilon, t\right)}\right\|_2^2$

Sampling Algorithm:
1. Set $x_T \sim \mathcal{N}(0, I)$ Then for $t = T, ..., 1$:
   1. $z \sim \mathcal{N}(0, I)$ if $t > 1$ else $z = 0$. (So output is stochastic)
   2. $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\overline{\alpha_t}}}\varepsilon_{\theta(x_t,t)}\right) + \sigma_t z$

U-net often used to estimate ε_θ since (up/de)sampling works well with noisy data where compressed and original spatial data are both important.

20 Restricted Boltzmann Machines



20.1 RBM Energy

$E(v, h) = -\sum_{i=1}^m \sum_{j=1}^n v_i W_{ij} h_j - \sum_{i=1}^m b_i v_i - \sum_{j=1}^n c_j h_j$ or rewritten $E(v, h) = -vWh^\top - bv^\top - ch^\top$ where $W \in \mathbb{R}^{m \times n}$
Discordance cost: $-vWh^\top$. Operating cost: $-bv^\top - ch^\top$

RBM vs Hopfield: Both find weights that minimize energy, and network converges to low-energy states, but Hopfield sets nodes to target pattern but RBM only visible nodes.

20.2 Energy Gap: (Like gradient but for binary nodes)

$\Delta E(v_i) = E(v_i = 1) - E(v_i = 0)$
$\Delta E(v_i) = -\sum_{j=1}^n W_{ij} h_j - b_i$, $\Delta E(h_j) = -\sum_{i=1}^m v_i W_{ij} - c_j$.
If $\Delta E(v_i) > 0$ then turn v_i

Networks can get stuck in sub-optimal state ⇒ Use stochastic neurons:
$P(h_j = 1 \mid v) = \sigma\left(\sum_i v_i W_{ij} + c_j\right)$, $P(v_i = 1 \mid h) = \sigma\left(\sum_i W_{ij} h_j + b_i\right)$, $\sigma(z) = \frac{1}{1+e^{-z/T}}$ T is temperature

Sampling Algorithm:
- Compute $p_i = P(v_i = 1 \mid h)$. For $i = 1, ..., m$:
  - Draw $r \sim \text{Uniform}(0, 1)$. If $p_i > r$ set $v_i = 1$, else set $v_i = 0$

If network run freely, states will be visited with probability $q(v, h) = \frac{1}{Z}e^{-E(v,h)}$ with partition function $Z = \sum_{v,h} e^{-E(v,h)}$.
Since lower-energy states are visited more frequently, then $E(v^{(1)}, h^{(1)}) < E(v^{(2)}, h^{(2)}) \Rightarrow q(v^{(1)}, h^{(1)}) > q(v^{(2)}, h^{(2)})$

20.3 Generation

Suppose inputs $v \sim p(v)$, we want RBM to act like generative model q_θ such that $\max_\theta \mathbb{E}_{v \sim p}\left[\ln q_{\theta(v)}\right]$

$$L = -\ln q_{\theta(V)} - \ln\left(\frac{1}{Z}\sum_h e^{-E_{\theta(V,h)}}\right)$$

Let loss be $= -\ln\left(\sum_h e^{-E_{\theta(V,h)}}\right) + \ln\left(\sum_v \sum_h e^{-E_{\theta(v,h)}}\right)$, which can be decomposed into $L = L_1 + L_2$

$$\nabla_\theta L_1 = -\nabla_\theta \frac{\sum_h e^{-E_{\theta(V,h)}}}{\sum_h e^{-E_{\theta(V,h)}}} = \sum_h \frac{e^{-E_{\theta(V,h)}}}{\sum_h e^{-E_{\theta(V,h)}}} \nabla_\theta E_\theta(V, h)$$

Then, since the fraction above is equivalent to $q_{\theta(h \mid V)}$, we write $\nabla_\theta L_1 = \sum_h q_{\theta(h \mid V)} \nabla_\theta E_{\theta(V,h)} = \mathbb{E}_{q(h \mid V)}\left[\nabla_\theta E_{\theta(V,h)}\right]$

$$\nabla_\theta L_2 = -\frac{\sum_{v,h} e^{-E_\theta} \nabla_\theta E_\theta}{\sum_{v,h} e^{-E_\theta}} = -\sum_{v,h} q_{\theta(v,h)} \nabla_\theta E_\theta(v, h)$$
$$= -\mathbb{E}_{q(v,h)}\left[\nabla_\theta E_{\theta(v,h)}\right]$$

$\nabla_{W_{ij}} E(V, h) = -V_i h_j$ and $\nabla_{W_{ij}} E(v, h) = -v_i h_j$ and thus $\nabla_{W_{ij}} L = -\mathbb{E}_{q(h \mid V)}[V_i h_j] + \mathbb{E}_{q(v,h)}[v_i, h_j]$.
First term represents expected value under posterior distribution, and second term under joint distribution

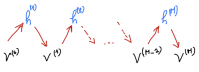
20.4 Contrastive Divergence for Training

1. Clamp visible states to V and calculate hidden probabilities $q(h_j \mid V) = \sigma(VW_{,j} + c_j)$ and $\nabla_W L_1 = -V^\top \sigma(VW + c)$ which results in a rank-1 outer product in $\mathbb{R}^{m \times n}$
2. Compute expectation using Gibbs sampling. $\langle v_i h_j \rangle_{q(v,h)} = \sum_v \sum_h q(v, h) v_i h_j$. Practically, single network state is used to approximate expectation. Gibbs sampling is used to run the network freely and compute average $v_i h_j$, which has $\nabla_W L_2 = v^\top \sigma(vW + c)$.

Procedure for training: Given V calculate h, then given new h calculate v, and from new v calculate second h.

Weight update rule is: $W \leftarrow W - \eta(\nabla_W L_1 + \nabla_W L_2)$ where η is learning rate.

20.5 Sampling



After training, we can generate new M data points $V^{(1)}, V^{(2)}, ...V^{(M)}$ by performing Gibbs sampling from the conditional probabilities $P(v|h)$ as illustrated.

21 Adversarial Attacks

Consider classifier f that produces probability vectors.

Classificaiton errors are defined as $R(f) \triangleq \mathbb{E}_{(x,t)\sim D}[\text{card}\{\arg\max_i y_i \neq t \mid y = f(x)\}]$

ε-Ball: $\text{B}(x, \varepsilon) = \{x' \in X \mid \|x - x'\| \leq \varepsilon\}$

Adversarial Attack: Find $x' \in \text{B}(x, \varepsilon)$ such that $\arg\max_{i(y_i)} \neq t$ for $y = f(x')$

Untargeted Gradient-Based Whitebox Attack: $x' = x + k\nabla_x E(f(x; \theta), t(x))$ Targeted: $x' = x - k\nabla_x E(f(x; \theta), l)$

21.1 Fast Gradient Sign Method

Adjust each pixel by ε, so $\delta x = \varepsilon \ \text{sign}(\nabla_x E)$ (This is because it's non-differentiable, so model training can't adjust for it)

Minimal Perturbation: Smallest $\|\delta x\|$ causing misclassification: $\min_{\|\delta x\|}\left[\arg\max_{i(y_{i(x)})} \neq t(x)\right]$

22 Adversarial Defense

During training, add adversarial samples to dataset with proper classification.

22.1 TRADES

Model: $f : X \to \mathbb{R}$ Dataset: (X, T); $X \in \mathbb{X}, T \in \{-1, 1\}$
$\text{sign}(f(X))$ indicates class. Classification is correct if $f(X)T > 0$

Robust Loss: $\mathcal{R}_{\text{rob}}(f) = \mathbb{E}_{X,T}[\text{card}\{X' \in \text{B}(X, \varepsilon) \mid f(X')T \leq 0\}]$ (Even if proper classification, if ε-ball can be fooled, it counts for loss)

Differentiable Training Loss: $\mathcal{R}_{\text{learning}} = \min_f \mathbb{E}_{(X,T)}[g(f(X))T]$ where g is smooth function.

Robust model optimizes over $\min_f \mathbb{E}_{(X,T)}\left[g(f(X)T + \max_{X' \in \text{B}(X,\varepsilon)} g(f(X)f(X'))\right]$ Term 1 ensures proper classification. Term 2 adds penalty for attacks.

Procedure:
1. For each gradient update:
   1. Run several gradient ascents to find X'
   2. Evaluate joint loss $g(f(X)T) + \beta g(f(X)f(X'))$
   3. Update weights off loss

23 Population Coding

The ability to encode data in a shape we want, and break apart a Black-Box NN to separate interpretable NN features that we can place in sequence.

Given activities of a neural network we can reconstruct input based, off of the specific activation values. Since decoding is linear function, this is regression, with MSE loss, so we can optimize for $\frac{1}{2P}\min_D \|Hd - X\|_2^2$, where $H^{(i)}$ is a row corresponding to activations of hidden layer.

The optimal linear decoding can be solved to be $d^* = (H^\top H)^{-1} H^\top X$.

This can be problematic if $H^\top H$ is poorly conditioned (almost singular), so instead we add noise to H, and get:

$$\|(H + \varepsilon)D - T\|_2^2 = \|(HD - T) + \varepsilon D\|_2^2$$
$$= (HD - T)^\top (HD - T) + 2(HD - T)^\top (\varepsilon D)$$
$$+ D^\top \varepsilon^T \varepsilon D$$

Since middle term is usually zero, since ε independent of $HD - T$, then if $\varepsilon^\top \varepsilon \approx \sigma^2 I$, then it is finally $\|HD - T\|_2^2 + \sigma^2 \|D\|_2^2$

We can also expand population coding to reconstruct vectors. We solve the matrix $D^* = \arg\min_D (\text{norm}\ HD - T)_F^2$ and T is a matrix where each row corresponds to a horizontal sample vector input.

24 Transformations

Population coding can pass data between neuron populations by transforming hidden activations.
Naive: Decode to data space and then re-encode.
Better: Bypass data space by multiplying decoder and encoder weights directly, resulting in rank-1 matrix. $W = D_{xy} E_B \in \mathbb{R}^{N \times M}$
$D_{xy} \in \mathbb{R}^{N \times 1}$ and $E_B \in \mathbb{R}^{1 \times M}$.
Using separate decoder-encoder matrices is computationally efficient. Total Time $O(N + M)$ for calculating AD and $(AD)E$ unlike tied weights taking $O(NM)$

25 Dynamics

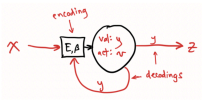
To build dynamic, recurrent networks via population coding methods, we'll leverage a dynamic model of LIF neurons based on current and activity. Our population coding methods will operate on activity, so we can modify the input current to establish forces on the activity.

$$\tau_s \frac{ds}{dt} = -s + C \ \text{Current}$$

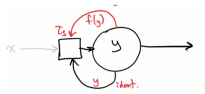
$$\tau_m \frac{dv}{dt} = -v + \sigma(s) \ \text{Activity}$$

Equilibrium values are when differential values equal 0

If $\tau_m \ll \tau_s$ then Activity function reaches equilibrium value, while current is still in dynamic state. Same for if $\tau_s \ll \tau_m$



If we're integrating the input, then $\tau_s \frac{ds}{dt} = -s + \sigma(s)W + \beta + C$ a recurrent network. Then, $\frac{ds}{dt} = \frac{-s + v W^+ \beta}{\tau_s} + \tilde{\tau}$ where $\tilde{C} = xE$ because it is the re-integrated input, and isn't dependent on time constant.



Since we have multiple different forms of the dynamic system, we can generalize to $\frac{dy}{dt} = f(y)$

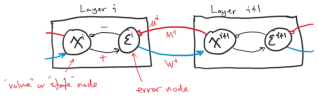
26 Biological Backprop

In neurons, updates can only use local information, but backpropagation updates to weights involves various layers of information.

26.1 Predictive Coding (PC)

- Predictions are sent down the hierarchy
- Errors are sent up the hierarchy

In a PC-Node, there are two parts: Value/State Node and an error node.



$\mu^i = \sigma((x^{i+1})M^i + \beta^i)$, this is our prediction. For simplicity, assume $W^i = (M^i)^T$

26.1.1 Error Node

$\tau \frac{d\varepsilon^i}{dt} = x^i - \mu^i - v_{\text{leak}}^i \varepsilon^i$, which at equilibrium reaches $\varepsilon^i = \frac{1}{v^i}(x^i - \mu^i)$

26.1.2 Generative Network

Given dataset x, y and $\theta = \{M^i, W^i\}_{i=1,...,L-1}$, the goal is to $\max_\theta \mathbb{E}_{(x,y)}[\log p(x \mid y)]$ $p(x \mid y) = p(x^1 \mid x^2)p(x^2 \mid x^3)...p(x^{L-1} \mid y) = p(x^1 \mid \mu^1)...p(x^{L-1} \mid \mu^{L-1})$

If we assume $x^i \sim \mathcal{N}(\mu^i, v^i)$, then $p(x^i \mid \mu^i) \propto \exp\left(-\frac{\|x^i - \mu^i\|^2}{2(v^i)^2}\right)$

Then, $-\log p(x^i \mid \mu^i) = \frac{1}{2}\left\|\frac{x^i - \mu^i}{v^i}\right\|^2 + C$, so as a result. $-\log p(x \mid y) = \frac{1}{2}\sum_{i=1}^{L-1} \left\|\varepsilon^i\right\|^2$

26.1.3 Hopfield Energy

$F = \frac{1}{2}\sum_{i=1}^{L-1} \left\|\varepsilon^i\right\|^2$ $\tau \frac{dx^i}{dt} = -\nabla_{x^i} F$

x^i shows up in two terms in F: $\varepsilon^i = x^i - \mu^{i(x^{i+1})} = x^i - (\sigma(x^{i+1})M^i + \beta^i)$ and $\varepsilon^{i-1} = x^{i-1} - \mu^{i-1}(x^i) = x^{i-1} - (\sigma(x^i)W^{i-1} + \beta^{i-1})$

Therefore $\nabla_{x^i} F = \varepsilon^i \frac{\partial \varepsilon^i}{\partial x^i} + \varepsilon^{i-1}\frac{\partial \varepsilon^{i-1}}{\partial x^i} = \varepsilon^i - \sigma'(x^i) \odot \left(\varepsilon^{i-1}(W^{i-1})^T\right)$

26.1.4 Dynamics

$\tau \frac{dx^i}{dt} = \sigma'(x^i) \odot (\varepsilon^{i-1}M^i) - \varepsilon^i$ and $\tau \frac{d\varepsilon^i}{dt} = x^i - \mu^i - v^i \varepsilon^i$. Then learning M^i involves $\nabla_{M^i} F = -(\sigma(x^{i+1}))^T \varepsilon^i$ with systems $\frac{dM^i}{dt} = \sigma\left(x^{(i+1)^T}\right)\varepsilon^i$ and $\tau \frac{dW^i}{dt} = (\varepsilon^i)^T \cdot \sigma(x^{i+1})$.

At equilibrium we get $\varepsilon^i = \sigma'(x^i) \odot \left[\varepsilon^{i-1}\left(W^{(i-1)^T}\right)\right]$ where $\frac{\partial F}{\partial \mu^i} = -\varepsilon^i$

Training: Clamp $x^1 = X$ and $x^L = Y$ and run to equilibrium. x^i, ε^i reach equilibrium quickly. Then use the two systems based on dM^i and dW^i to update M^i and W^i

Generating: Clamp $x^L = Y$ and run to equilibrium and x^1 is a generated sample

Inference: Clamp $x^1 = X$ run to equilibrium and $\arg\max_j(x_j^L)$ is the class

This work overcomes the local learning condition because running to equilibrium allows information to spread through the net.

27 Generative Adversarial Networks

Two networks: Generative Network and Discriminative Network

$D(x; \theta)$ - Probability x is real. $G(z; \varphi)$ - Create input sample from random noise z drawn from p_z distribution.

Loss function: $E(\Theta, \varphi) = \mathbb{E}_{x \sim \mathcal{R}, \mathcal{F}}[\mathcal{L}(D(x; \theta), t)] + \mathbb{E}_{z \sim p_z}[\mathcal{L}(D(G(z; \varphi), \varphi), 1)]$ Term 1: Minimize θ to make discriminator better Term 2: Minimize φ to make generator better at producing fake inputs.

Train discriminator: $\min_\theta \mathbb{E}_{\mathcal{R}, \mathcal{F}}[\mathcal{L}(y, t)]$ R are real inputs, F are fake. Update rule: $\theta \leftarrow \theta - \kappa\nabla_\theta \mathcal{L}(y, t)$ Train generator: $\min_\theta \mathbb{E}_{\mathcal{F}}[\mathcal{L}(y, 1)]$ Update rule: $\varphi \leftarrow \varphi - \kappa\nabla_\varphi \mathcal{L}(y, 1)$ (We use 1 to simulate a targeted adversarial attack with target 1) Gradients propagate through D down to G. Note that y is the discriminator being run on generated outputs.

28 Transformers

Sequential data inputs are first tokenized into vector embeddings. The input X is transformed into queries (input values) $Q = XW^{(Q)}$, Keys (tags) $K = XW^{(K)}$, and values (desired data) $V + XW^{(V)}$ using weight matrices $W^{(\cdot)} \in \mathbb{R}^{d \times l}$

Self-attention scores are calculated $S = QK^\top$ where $S_{ij} = q_i \cdot k_j$ scores query i against key j. Softmax is applied row-wise to $\frac{S}{\sqrt{d}}$ to get attention scores $A \in \mathbb{R}^{n \times n}$. Output of attention head is $A \cdot V = H \in \mathbb{R}^{n \times l}$

Positional encodings: $PE(i)_{2j} = \sin\left(\frac{i}{10000^{\frac{2j}{d}}}\right)$, $PE(i)_{2j+1} = \cos\left(\frac{i}{10000^{\frac{2j}{d}}}\right)$ freq changes with dimension j, and 10000 is scaling constant for large max seq len. Then, $\vec{x_i}' = \vec{x_i} + PE(i)$

Multi-Head Attention uses separate attention mechanisms to learn different features, then outputs are concatenated and linearized for output.