1 General Formula

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

L1 Norm $\sum_i |\theta_i|$

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

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

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

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

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

2 Neurons

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

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

Membrane Potential Difference in voltage across membrane. At rest, value is –70mV, enforced by Na-K Pump.

Action Potential Electrical impulse travelling along axon to the synapse.

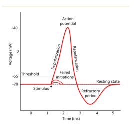
2.1 Hodgkin-Huxley Model

Model of Neuron based on Ion Channel components.

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

Gate dynamic system: $a(t) = \frac{da}{dt} = \frac{1}{\tau_{a(V)}}\left(a_{\infty(V)} - a\right)$ (For all gates- Replace a, with n, h, t)

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 $I = C\frac{dV}{dt}$: 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

Action Potential form: 

Process: Stimulus breaks threshold causing Na+ channels to open, then close at action potential. Potassium channels open at action potential and close at refractory period.

2.2 Leaky-Integrate-and-Fire Model

Spike shape is constant over time, more important to know when spiked than shape. LIF only considers sub-threshold voltage and when peaked.

Dynamics system: $c\frac{dV}{dt} = J_{in} - g_{L(V-V_L)} \Rightarrow \tau_m \frac{dV}{dt} = V_{in} - (V - V_L)$ for $V < V_{th}$

This is dimensioned. Dimensionless converts $v_{in} = \frac{v_{in}}{v_{th} - V_L}$ to become $\tau_m \frac{dv}{dt} = v_{in} - v$

Then spike occurs when $v = 1$ and we set a refractory period of t_{ref} before starting at 0 again.

Explicit Model: $v(t) = v_{in}\left(1 - e^{-\frac{t}{\tau}}\right)$

Firing Rate: $\frac{1}{\tau_{\text{ref}} - \tau_m \ln(1 - 1v_\in)}$ for $v_\in > 1$ **Tuning Curve**: Graph showing how neuron reacts to different input currents.

2.3 Activation Functions (Sigmoidal)

Logistic Curve: $\frac{1}{1+e^{-z}}$, $\arctan(h)$, $\tanh(z)$, threshold, ReLU, Softplus: $\log(1 + e^z)$, SoftMax: $\frac{\exp(z_i)}{\sum_j \exp(z_j)}$, ArgMax

3 Synapses

In real neuron, **Presynaptic action potential** releases **neurotransmitters** across **synaptic cleft** which binds to **postsynaptic** receptors

Equation for current entering **postsynaptic neuron**: $h(t) = \begin{cases} kt^n e^{-\frac{t}{\tau_s}} \text{ if } t \geq 0 (\text{for some } n \in \mathbb{Z}_{\geq 0}) \\ 0 \text{ if } t < 0 \end{cases}$

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

Postsynaptic potential filter: $h(t)$

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: $s(t) = a(t) * h(t)$ For each spike in spike train, run the postsynaptic potential filter on it, then sum each spike. Essentially, convolve the spike train to the postsynaptic potential filter.

Derivation:

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

3.1 Neuron Activities

Neurons have multiple connections, with different strengths (weights). We can represent weights between neuron layers via weight matrix: $W \in R^{N \times M}$

Compute neuron function as: $\vec{z} = \vec{x}W + \vec{b}$, thus $\vec{y} = \sigma(\vec{z})$

3.1.1 Bias Representation

Add a neuron with constant value 1 for each layer, and use its weights as biases.

$\vec{x}W + \vec{b} = (\vec{x} \;\; 1) \cdot \begin{pmatrix} W \\ \vec{b} \end{pmatrix}$

3.1.2 Connections between spiking Neurons

Let $n = 0$ for simplicity for $h(t)$, then it is a solution of $\tau_s \frac{ds}{dt} = -s$, $s(0) = \frac{1}{\tau_s}$

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

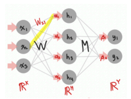
4 Gradient Descent

Our goal is to minimize $E(\theta)$ (expected Error), so we define gradient $\nabla_\theta E = \left(\frac{\partial E}{\partial \theta_1} \; \cdots \; \frac{\partial E}{\partial \theta_n}\right)$

Error gradient: $\tau \frac{d\theta}{dt} = \nabla E(\theta)$, t is a parameter for "time" as we move through parameter space. Euler step: $\theta_{n+1} = \theta_n + k\nabla E(\theta_n)$

5 Error Backpropagation

$\nabla_{z^{(l+1)}} E = \frac{\partial E}{\partial z^{(l+1)}}$ $h^{(l+1)} = \sigma(z^{(l+1)}) = \sigma(W^{(l)}h^{(l)} + b^{(l+1)})$ Basically, h is hidden layer, z is input current, W is weight matrix, b is bias. $\nabla_{z^{(l)}} E = \frac{dh^{(l)}}{dz^{(l)}} \odot \left[\nabla_{z^{(l+1)}E \cdot (W^{(l)T})}\right]$ 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, $\nabla_{z^{(l)}} E = \sigma'(z^{(l)}) \odot \left(\nabla_{z^{(l+1)}} E \cdot (W^{(l)})^T\right)$ $\nabla_{W^{(l)}} E = [h^{(l)}]^T \nabla_{z^{(l+1)}} E$

5.1 Vectorization

We can generalize this process to take a batch of samples by letting x be a matrix of samples instead of just one sample. Then, note $\nabla_{z^{(l)}} E$ is a matrix with same dimension as $z^{(l)}$ as desired. Further, note that $\nabla_{W^{(l)}} E$ is a gradient vector that sums the weight gradient matrix from each sample.

6 Auto-Differentiation

6.1 Expression Graph

Each operation is a square with its variable dependencies. Each variable has a pointer to its creator, which is the operation that created it.

6.1.1 Pseudocode

Function $f = g(x, y, ...)$
- Create g Op object
- Save references to args $x, y, ...$
- Create Var for output f
- Set g.val as $g(x, y, ...)$
- Set f.creator to the g Op.

6.2 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 $\cdot \frac{\partial y}{\partial x}$ Also add wherever multiple branches converge, as is normal in derivatives calculation.

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

7 Neural Nets w/w Auto-Diff

7.1 Gradient Descent Pseudocode

- Initialize v, κ
- Make expression graph for E
- Until convergence:
  - Evaluate E at v
  - Zero-grad
  - Calculate gradients
  - Update $v \leftarrow v - \kappa \text{ v.grad}$

7.2 Neural Learning

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

7.2.1 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()
- $\theta \leftarrow \kappa \cdot \theta.\text{grad}$

8 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 $E(Y, T) \approx E(\tilde{Y}, \tilde{T}) = \frac{1}{B}\sum_B L(y_{\gamma_b}, t_{\gamma_b})$

Update after each batch B, and then continue gathering batches until all of dataset has been sampled to complete epoch.

8.1 Momentum

Use: Gradient descent oscillates often reaching optimum; grad.desc stops at local optimum (not global). 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 $w^{(t)} \leftarrow w^{(t-1)} - \eta v^{(t)}$

We think of v_n as the "velocity" and A_n (gradient vector) is the force. This gives direction and magnitude. If we move in the same direction all the time, then we gather "momentum", and are able to make bigger steps, by increasing velocity.

9 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.

In logistic activation functions, $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ which has maximum value 0.25, so when you chain activation functions, they shrink by at least a factor of 4 each layer, so the closer the layer to the input, the smaller its 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.

10 Convolutional Neural Networks

Convolution A mathematical operation that combines two functions by sliding them against each other. Essentially, you create a window, where each element in the window is a function, then you transform the original function over the window to get a frankenstein function.

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, which creates a tensor for the next layer. We can have multiple kernels per input layer.

Stride The amount the kernel is shifted against the input layer.

Padding Padding border of input layer with 0s to create a layer that is identical size as layer.

Note: The bias is attributed to the kernel as a hole, rather than the neurons in the kernel.

Note: 1x1 convolution layers are useful in multi-layered input. (Ex 1x1x64)

11 Batch Normalization

Some input features will have dramatically different magnitudes than others. But we'd need to accomodate the smallest magnitude feature to make meaningful steps, which slows down the larger magnitude rate.

The goal is to rescale all hidden layer outputs into normalized values.

Use the standard formulas for means and variance for each hidden layer output, then we rescale the inputs into the next layer as:

$\hat{h_i}^{(d)} = \frac{h_i^{(d)} - \mu_i}{\sigma_i}$ or $\hat{h_i}^{(d)} = \frac{h_i^{(d)} - \mu_i}{\sqrt{\sigma_i^2 = \varepsilon}}$ for small $\varepsilon > 0$ for if variance is 0, to prevent instabilities.

Then we rescale output with learnable parameters γ_i, β_i

Following process: [Hidden layer i] -> Normalization -> Rescaling -> [Hidden Layer i+1]

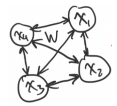
$y_i^{(d)} = \gamma_i \hat{h_i}^{(d)} + \beta_i$

Batch normalization affects convergence rate for learning quickly. The reason is unknown but there are hypotheses:
- Mitigates vanishing/exploding gradients
- Guards against internal covariate shift (shallow layers (near output) learn quicker than deep layer, so whenever deep layers learn, they invalidate the outputs of the shallow layer, who has to learn again. By normalizing, the inputs remain relatively stable, so shallower layers aren't as "affected" by differences from deep layer changes.

12 Hopfield Networks

Content-Addressable Memory System that can take part of a pattern and fill in the most likely matches from memory.

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



Each x_i is assigned: -1 if $\vec{x}W + b_i < 0$ and 1 if $\vec{x}W + b_i \geq$

Since the graph has cycles, backpropagation won't work. Instead we need to utilize the Hopfield Energy:

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

and diagonals of W are 0.

To minimize energy, we use gradient descent: $\frac{\partial E}{\partial x_j} = -\sum_{i \neq j} x_i W_{ij} - b_j$ or $\nabla_{\vec{x}} E = -\vec{x}W - \vec{b} \Rightarrow \tau_x \frac{d\vec{x}}{dt} = \vec{x}W + \vec{b}$

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

If $i = j$ $\frac{\partial E}{\partial W_{ii}} = -x_i^2 = -1$

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

Over M targets we have: $\nabla_W E = -\frac{1}{M}\sum_{s=1}^M (\vec{x}^{(s)})^T \vec{x}^{(s)} + I = -\frac{1}{M}X^T X + I$

Thus $\tau_w \frac{dW}{dt} = \frac{1}{M}X^T X - I$

13 Recurrent Neural Networks

Hidden layer can encode input sequence, allowing sequential data to be outputted.

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: $h^i = \sigma(x^i U + h^{i-1}W + b)$ $y = \sigma(h^i V + c)$

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

Following gradients: $\nabla_{z^k} E = \nabla_{z^k}\left(\sum_{i=1}^t L(y^i, t^i)\right) = \sum_{i=1}^t \nabla_{z^k} L(y^i, t^i) = \nabla_{z^k} L(y^k, t^k) = \nabla_{y^k} L(y^k, t^k) \odot \sigma'(z^i)$

$\nabla_V E = \sum_{i=1}^T h^i \nabla_{z^k} L(y^i, t^i)$

In order to derive hidden layer gradients, define $E^k = \sum_{i=k}^T L(y^i, t^i)$

Since h_k only affects h_m where $m \geq k$ then $\nabla_{h^k} E = \nabla_{h^k} E^k$

$\nabla_{h_\tau} E = \nabla_{z^\tau} E^\tau \frac{\partial z^\tau}{\partial h^\tau} = \nabla_{z^\tau} E^\tau)V^T$

Then, we can compute $\nabla_{h^i} E$ recursively.

$\nabla_{h^i} E = (\nabla_{h^{i+1}} E^{i+1} \odot \sigma'(s^{i+1}))W^T + (\nabla_{y^i} L(y^i, t^i) \odot \sigma'(z^i))V^T$

Once you have $\nabla_{h^i} E$ you can compute gradient with respect to deeper weights and biases recursively.

14 Gated Recurrent Units

14.1 Problem

Vanilla RNNs struggle to capture long-range dependencies, because of the vanishing gradient problem, as gradients multiply over long ranges, meaning the relative effect of distant tokens to the current token are minimal.

Consider a simple example:

$h^n = wh^{n-1} + x^n$, if we unroll this equation, we get $h^n = w^{n-1}h_1 + w^{n-1}x^1 + w^{n-2}x^2 + ...$

If $|w| < 1$ then w^n shrinks exponentially as n grows, meanign earlier information has exponentially less influence on h^n over time. If $|w| > 1$ then h^n magnitude grows exponentially, making training unstable.

14.2 Gated Recurrent Units

Use gating mechanisms to control which information to retain.

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

Gate Mechanism: $\overrightarrow{g^n} = \sigma\left(\overrightarrow{h^{n-1}}W_g + \overrightarrow{x^n}U_g + \vec{b}_g\right)$ The sigmoid ensures $g_i^n \in (0, 1)$ meaning it's a soft switch.

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

$\overrightarrow{g^n}$ 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.

14.3 GRU Network

The GRU extends from RNN by adding two gating mechanisms, update gate and reset gate (same format as gate mechanism above).

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

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

Reset gate tries to "forget" previous hidden state information for new candidate. When $\vec{r^n}$ is close to 0, past is discarded, making it more focused on recent information, when close to 1, then more of hidden state is retained.

14.4 minGRU

For a sequence of length N, we'd need to compute each component sequentially, nullifying parallelization advantages from GPUs.

minGRU simplifies to: $\overrightarrow{g^n} = \sigma\left(\overrightarrow{x^n}U_g + \vec{b}_g\right)$ $\overrightarrow{\widetilde{h^n}} = \overrightarrow{x^n}U + \vec{b}$ $\overrightarrow{h^n} = \overrightarrow{g^n} \odot \overrightarrow{\widetilde{h^n}} + (1 - \overrightarrow{g^n}) \odot \overrightarrow{h^{n-1}}$

minGRU allows $\overrightarrow{g^n}$ and $\overrightarrow{\widetilde{h^n}}$ to be parallelizable. Hidden states can use parallel scan to compute values in $\mathrm{O}(\log N)$ instead of $\mathrm{O}(N)$ sequentially.

15 Autoencoders

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

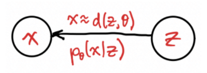
The model is trained using loss function minimizing $L(x', x)$ where x' is reconstructed input and x is original input.

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

16 Variational Autoencoders

Goal: Create autoencoder that generates reasonable samples not in training set. We encode a latent probability distribution, instead of encoding.

x is input, z is latent space, so encode $p(z)$. Decoding is $d(z, \theta)$. θ are decoder weights. Want to maximize $p(x) = \int p_{\theta(x \mid z)} p(z) dz$—the decoding probability.



Assume $p(x \mid z)$ is Gaussian (for simplicity), then NLL is $-\ln p_{\theta(x \mid z)} = \frac{1}{2\Sigma^2}\|X - d(z,\theta)\|^2 + C$ which is what we want to maximize. Our goal is $\min_\theta \mathbb{E}_{z\sim p(z)}[\|X - d(z,\theta)\|^2]$, and $\mathbb{E}_{p(z)}\left[p_{\theta(x \mid z)}\right] = \int p_{\theta(x \mid z)} p(z) dz$. But we don't know how to sample $p(z)$, so we assume a distribution $q(z)$ to approximate value.

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

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

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

We will then choose $p(z) \sim \mathcal{N}(0, I))$ and want to $\min_q KL(q(z))\|\mathcal{N}(0, I))$. Our encoder will be designed to have outputs μ, σ which aren't actual means, or standard deviations, but are just parameters for the distribution.

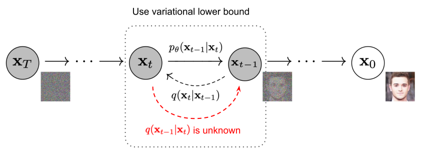
These gaussians result in $KL(\mathcal{N}(\mu, \sigma))\|\mathcal{N}(0, I)) = \frac{1}{2}(\sigma^2 + \mu^2 - \ln(\sigma^2) - 1)$

The other part of objective, $\mathbb{E}_q[\ln p(x \mid z)]$ 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 is a reparameterszation trick done so we can backpropagate on x, θ, by making the ε a separate noise vector.

Procedure: Encode x by finding $\mu(x, \theta)$ and $\sigma(x, \theta)$. Sample $z = \mu + \varepsilon\sigma, \varepsilon = \mathcal{N}(0, I)$. Calc KL Loss. Decode $\hat{x}$ using decoder. Calc reconstruction loss. MSE for Gaussian $p(x \mid \hat{x})$; BCE for Bernoulli. Gradient descent on θ on combined loss, with β parameters on KL divergence, to adjust importance.

17 Diffusion Models

Generative model using noise as seeds for making new images, by training on a model that adds noise to samples so it can then reverse the process.



Use variational lower bound
$\mathbf{x}_T$
$\mathbf{x}_t$
$\mathbf{x}_{t-1}$
$\mathbf{x}_0$
$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$
$q(\mathbf{x}_t|\mathbf{x}_{t-1})$
$q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is unknown

Forward process: $q(x_t \mid x_{t-1}) = \mathcal{N}\left(x_t; \sqrt{1-\beta_t} x_{t-1}, \beta_t I\right)$. 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.

If we unwrap x_t recursively, we get $x_t = \sqrt{\overline{\alpha_t}} x_0 + \sum_i c_i \varepsilon_i$ 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 gaussion, then $x_t = \sqrt{\overline{\alpha_t}} x_0 + \sqrt{1 - \overline{\alpha_t}} \varepsilon$.

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 $D_{KL}\left(q(x_{t-1}|x_t, x_0), p_{\theta(x_{t-1}|x_t)}\right)$ for $t = 2, ..., T$. $q(x_{t-1}|x_t, x_0) = \mathcal{N}\left(x_{t-1}|\mu_q, \Sigma_q\right), \mu_q = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}} \varepsilon_t\right)$

Further, parameterized probability density p_θ is assumed to be a Gaussian distribution, with the same format, but $\varepsilon_{\theta(x_t, t)}$ instead of ε_t

Then the KL-Divergence gets simplified 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:

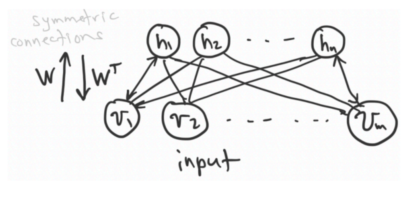
1. Take x_0 from dataset
2. $t \sim$ Uniform($\{1, ..., T\}$)
3. $\varepsilon \sim \mathcal{N}(0, I)$
4. Do gradient descent on simplified Loss function

Sampling Algorithm:

```
1. Set $x_T \sim \mathcal{N}(0, I)$
2. For $t = T, ..., 1$
   1. $z \sim \mathcal{N}(0, I)$ if $t > 1$ else $z = 0$ We add noise to make stochastic sample
   2. $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\overline{\alpha_t}}} \varepsilon_{\theta(x_t,t)}\right) + \sigma_t z$
3. Return $x_0$
```

We typically use a U-net to estimate ε_θ because upsampling/desampling works well with noisy data where compressed and original spatial data are both important.

18 Restricted Boltzmann Machines



Network consists of:

- Hidden layer and visible layers (nodes only binary) and connections between layers are symmetric by weight matrix W

18.1 RBM Energy

- RBM's energy is $E(v, h) = -\sum_{i=1}^m \sum_{j=1}^n v_i W_{ij} h_j - \sum_{i=1}^m b_i v_i - \sum_{j=1}^n c_j h_j$ 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 runninng network converges to low-energy states, but Hopfield sets nodes to target pattern but RBM only visible nodes.

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

$\delta E(v_i) = E(v_i = 1) - E(v_i = 0)$, for $i = 1, ..., m : \delta E(v_i) = -\sum_{j=1}^n W_{ij} h_j - b_i$ For $j = 1, ..., n : \delta E(h_j) = -\sum_{i=1}^m v_i W_{ij} - c_j$. Then if $\delta E(v_i) > 0$ then turn v_i off, meaning $E(v_i = 1) > E(v_i = 0)$

One issue is that these binary networks can get stuck in sub-optimal state, so we can use stochastic neurons as $P(h_j = 1 \mid v) = \sigma\left(\sum_i v_i W_{ij} + c_j\right)$ (similar for visible nodes) where logistic is defined as $\sigma(z) = \frac{1}{1+e^{-\frac{z}{T}}}$, which is temperature dependent.

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 $v_i = 0$

18.3 Network Dynamics

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

18.4 Generation

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

Let loss be $L = -\ln q_{\theta(V)}$ for given V, or $L = -\ln\left(\frac{1}{Z}\sum_h e^{-E_{\theta(V,h)}}\right) = -\ln\left(\sum_h e^{-E_{\theta(V,h)}}\right) + \ln\left(\sum_v \sum_h e^{-E_{\theta(v,h)}}\right)$, which can be decomposed into $L = L_1 + L_2$

18.5 Gradient of L_1

- To optimize parameter, we need to compute gradient of loss function:

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

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

18.6 Gradient of L_2

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

18.7 Gradient for W_{ij}

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

18.8 Contrastive Divergence for Training

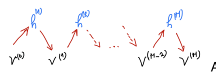
Step 1: Clamp visible states to V and calculate hidden probabilities $q\left(h_j \mid V\right) = \sigma\left(VW_{,j} + c_j\right)$ and $\nabla_W L_1 = -V^\top \sigma(VW + c)$ which results in a rank-1 outer product in $\mathbb{R}^{m\times n}$

Step 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.

18.9 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 below:

19 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: $\mathrm{B}(x, \varepsilon) = \{x' \in X \mid \|x - x'\| \leq \varepsilon\}$

Adversarial Attack: Find $x' \in \mathrm{B}(x, \varepsilon)$ 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: $x' = x - k\nabla_x E(f(x; \theta), l)$

19.1 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: $\min_{\|\delta x\|}\left[\arg\max_{i\left(y_{i(x)}\right)} \neq t(x)\right]$

20 Adversarial Defense

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

20.1 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: $\mathcal{R}_{\text{rob}}(f) = \mathbb{E}_{X,T}[\text{card}\{X' \in \mathrm{B}(X, \varepsilon) \mid f(X')T \leq 0\}]$ (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 $\min_f \mathbb{E}_{(X,T)}\left[g(f(X)T) + \max_{X'\in \mathrm{B}(X,\varepsilon)} g(f(X)f(X'))\right]$ 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
```

21 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 $d^* = \left(H^\top H\right)^{-1} H^\top X$.

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

$$\begin{aligned}\|(H + \varepsilon)D - T\|_2^2 = \|(HD - T) + \varepsilon D\|_2^2 \\ &= (HD - T)^\top(HD - T) + 2(HD - T)^\top(\varepsilon D) \\ &+ D^\top \varepsilon^T \varepsilon D\end{aligned}$$

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 $D^* = \arg\min_D (\texttt{norm}\ HD - T)_F^2$ (frobenius norm) and T is a matrix where each row corresponds to a horizontal sample vector input.

22 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 $O(NM)$

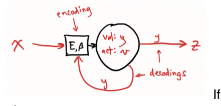
23 Dynamics

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.

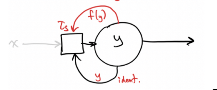
$$\tau_s \frac{ds}{dt} = -s + C \text{ Current}$$
$$\tau_m \frac{dv}{dt} = -v + \sigma(s) \text{ Activity}$$

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, $\frac{ds}{dt} = \frac{-s+vW+\beta}{\tau_s} + \tilde{\tau}$ 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 $\frac{dy}{dt} = f(y)$

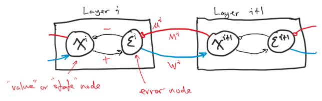
24 Biological Backprop

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

24.1 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 $W^i = (M^i)^T$

24.1.1 Error Node

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

24.1.2 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. $-\log p(x \mid y) = \frac{1}{2}\sum_{i=1}^{L-1} \|\varepsilon^i\|^2$

24.1.3 Hopfield Energy

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

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

24.1.4 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.

25 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.

26 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 $W^{(\_)} \in \mathbb{R}^{d\times l}$

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:$PE(i)_{2j} = \sin\left(\frac{i}{10000^{2\frac{j}{d}}}\right), PE(i)_{2j+1} = \cos\left(\frac{i}{10000^{2\frac{j}{d}}}\right)$ 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.