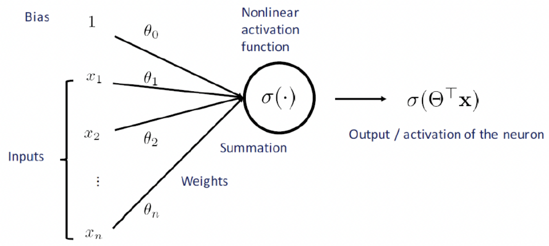
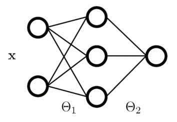
# 1 Deep neural networks

Deep learning learns multiple (hierarchical) layer of data representation (feature). Neural networks scale with compute, data/model size (vs. other ML approaches). Artificial neural networks is a simplified version of biological NN.



- Training dataset $\{(\boldsymbol{x}_1, y_1), \cdots (\boldsymbol{x}_n, y_n)\}$.
- NN $f(\boldsymbol{x}; \Theta) \in \mathbb{R}$ parameterized by $\Theta$.
- **Forward propagation** $\hat{y} = \sigma(\Theta_k^T \sigma(\cdots \sigma(\Theta_1^T \boldsymbol{x})))$



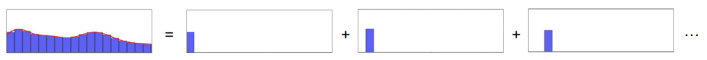
NNs with $\geq 2$ layers i.e. 1 hidden layer can model complex functions.

## 1.1 Universal approximation theorem

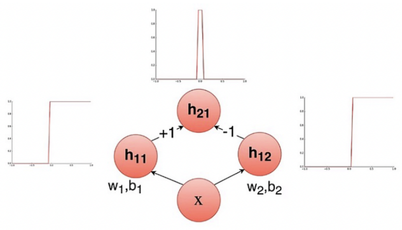
One hidden layer (with enough width) is enough to approximate all continuous functions.

### 1.1.1 Arbitrary width case

Let $C(X, \mathbb{R}^m)$ denote the set of continuous functions from a subset $X$ of a Euclidean $\mathbb{R}^n$ space to $\mathbb{R}^m$ space. Let $\sigma \in C(\mathbb{R}, \mathbb{R})$. Note that $(\sigma \circ x)_i = \sigma(x_i)$, so $\sigma \circ x$ denotes $\sigma$ applied to each component of $x$. Then $\sigma$ is polynomial if and only if $\forall n \in \mathbb{N}, m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n, f \in C(K, \mathbb{R}^m), \epsilon > 0$ there exist $k \in \mathbb{N}, A \in \mathbb{R}^{k \times n}, b \in \mathbb{R}^k, C \in \mathbb{R}^{m \times k}$ such that $\sup_{x \in K} \| f(x) - g(x) \| < \epsilon$ where $g(x) = C \circ (\sigma \odot (A \cdot x + b))$.



(a) We can approximate continuous function with piece-wise linear functions.



(b) Can construct by subtracting 2 step functions.

**Proof sketch**

### 1.1.2 Bounded depth and bounded width case

There exists an activation function $\sigma$ which is analytic, strictly increasing and sigmoidal and has the following property: For any $f \in C[0,1]^d$ and $\epsilon > 0$ there exists constant $d_i, c_{ij}, \theta_{ij}, \gamma_i$ and vectors $\boldsymbol{w}^{ij} \in \mathbb{R}^d$ for which

$$\left| f(\boldsymbol{x}) - \sum_{i=1}^{6d+3} d_i \sigma \left( \sum_{j=1}^{3d} c_{ij} \sigma(\boldsymbol{w}^{ij} \cdot x - \theta_{ij}) - \gamma_i \right) \right| < \epsilon$$

for all $\boldsymbol{x} = (x_1, \cdots, x_d) \in [0,1]^d$.

## 1.2 Training DNNs

**Objective**: find a parameter that minimizes error (or empirical risk)

$$\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} \ell(f(\boldsymbol{x}_i; \Theta), y_i) := L(\Theta) \qquad \Theta^{(t+1)} = \Theta^{(t)} - \gamma \nabla L(\Theta^{(t)})$$

where $\ell(\cdot, \cdot)$ is a loss function. **Gradient descent** (GD) updates parameters iteratively to the gradient direction.

### 1.2.1 Backpropagation

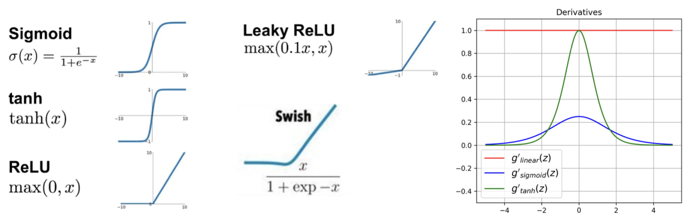
**TL;DR** Adjust the last layer weights $\Theta_k$. Propagate error back to each previous layers. Repeat for $\Theta_{k-1}, \cdots, \Theta_1$.

Consider the input $(\boldsymbol{x}_i, y_i)$. Forward propagation to compute $\hat{y}_i = f(\boldsymbol{x}_i; \Theta)$. $i$-th layer intermediate output $s_i = \Theta_i^T \boldsymbol{h}_{i-1}$. Compute MSE loss $\ell(\hat{y}_i, y_i) = 1/2\ (y_i - \hat{y}_i)^2 := E_i$.

$$\frac{\partial E_i}{\partial \hat{y}_i} = \frac{\partial}{\partial \hat{y}_i} \frac{1}{2} (y_i - \hat{y}_i)^2 = -(y_i - \hat{y}_i)$$

$$\frac{\partial E_i}{\partial s_k} = \frac{\partial E_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial s_k} = \frac{\partial E_i}{\partial \hat{y}_i} \frac{\partial}{\partial s_k} \sigma(s_k) = (y_i - \hat{y}_i) \sigma'(s_k)$$

### 1.2.2 Activation functions

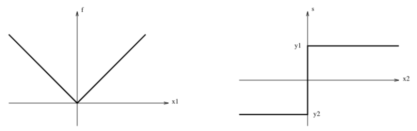


$\tanh(x) = 2\sigma(2x) - 1$ and $d\tanh(x)/dx = 1 - \tanh^2(x)$

### 1.2.3 Subdifferential

Set of all subgradients of $f$ at $x$ is called the **subdifferential** of $f$ at $x$, written $\partial f(x)$ if

- $\partial f(x)$ is a closed set
- $\partial f(x)$ nonempty (if $f$ convex, and finite near $x$)
- $\partial f(x) = \{\nabla f(x)\}$ if $f$ is differentiable at $x$
- if $\partial f(x) = \{g\}$, then $f$ is differentiable at $x$ and $g = \nabla f(x)$



**The absolute value function (left), and its subdifferential (right)**

In many cases, don't need complete $\partial f(x)$; sufficient to find one $g \in \partial f(x)$.

### 1.2.4 GD in practice

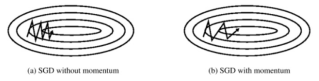
| | Batch | **Stochastic** |
|---|---|---|
| Calculate gradients with | All data | Subset of data |
| Computation | Heavy | Less |
| Convergence | Quick | Long |
| Avoid local optimum | Hard | Sometimes |

- Standardization can be helpful; Increase convergence speed. e.g. Max-min or z-score normalization
- Mini-batch training + standardization can be a good option.

### 1.2.5 Optimizers

**Momentum** accelerates GD when we have surface that curves more steeply in one direction than in another.

$$\theta = \theta - \eta \nabla J(\theta; x, y) \qquad \begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla J(\theta; x, y) \\ \theta &= \theta - v_t \end{aligned}$$



**Momentum dampens the oscillation.**

**Adam** calculates individual adaptive learning rate for each parameter from estimates of first and second moments of gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \qquad \text{(Biased corrected estimates)}$$

$$\theta_{t+1} = \theta_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \qquad \text{(Parameter update)}$$

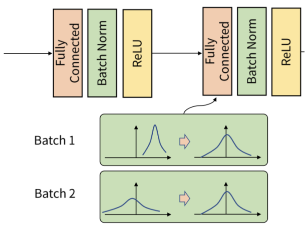
### 1.2.6 Parameter initialization

$W \sim N(0, \mathrm{Var}(W))$ where

$$\mathrm{Var}(W) = \begin{cases} \sqrt{1/n_{\text{in}}} & \text{(LeCun normal init.)} \\ \sqrt{2/(n_{\text{in}} + n_{\text{out}})} & \text{(Xavier normal init.)} \\ \sqrt{2/n_{\text{out}}} & \text{(He normal init.)} \end{cases}$$

### 1.2.7 Batch normalization

- Allow higher LR and reduce strong dependence on initialization.
- Activations have different distributions. BN makes them similar.
- <u>After</u> FC/Conv layer and <u>before</u> non-linearity layer.



**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

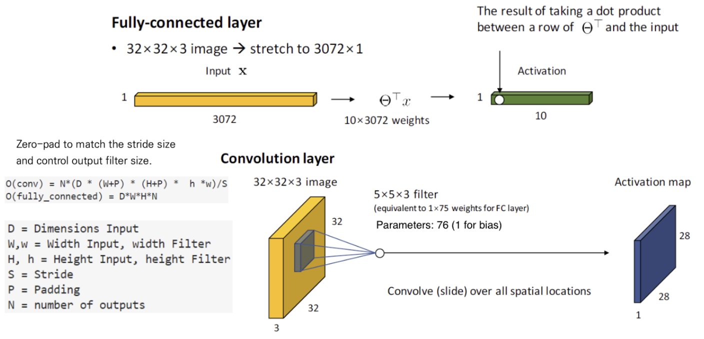
$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

## 1.3 Convolutional neural networks

- **Convolution**: Weight sharing and local connectivity
  - + **Translation invariance**
  - + Reduce the number of parameters (less overfitting)
  - + Learn local features
- **Pooling** (subsampling): operates on each activation map independently
  - + Translation invariance ↑ (to small transformations), Regularization
  - + Reduce the number of parameters and computation
- ConvNet is sequence of conv layers followed by non-linearity.


Fully-connected layer
• 32×32×3 image → stretch to 3072×1
Input x
1
3072
$\Theta^\top x$
10×3072 weights
The result of taking a dot product between a row of $\Theta^\top$ and the input
Activation
1
10
Zero-pad to match the stride size and control output filter size.
Convolution layer
32×32×3 image
32
32
3
5×5×3 filter
(equivalent to 1×75 weights for FC layer)
Parameters: 76 (1 for bias)
Convolve (slide) over all spatial locations
Activation map
28
28
1

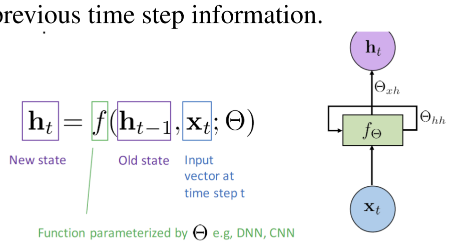
```
O(conv) = N*(D * (W+P) * (H+P) *  h *w)/S
O(fully_connected) = D*W*H*N
```

```
D = Dimensions Input
W,w = Width Input, width Filter
H, h = Height Input, height Filter
S = Stride
P = Padding
N = number of outputs
```

## 1.4 Recurrent neural networks

Markov chain $Pr(w_{i+1}|w_i)$. Language model $Pr(w_{i+1}|w_1, \cdots, w_i)$ becomes a large-scale classification task at every time $i$ since its vocabulary size is large.

RNN models temporal information. Hidden states as a function of inputs and previous time step information.


$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \Theta)$
New state
Old state
Input vector at time step t
Function parameterized by $\Theta$ e.g, DNN, CNN
$\mathbf{h}_t$
$\Theta_{xh}$
$\Theta_{hh}$
$f_\Theta$
$\mathbf{x}_t$

**Process sequence of vectors by applying recurrence formula at every** $t$

In simple RNN (or **vanilla** RNN), the state consists of single hidden vector. Recurrence formula becomes $\boldsymbol{h}_t = \tanh(\Theta_{hh}\boldsymbol{h}_{t-1} + \Theta_{xh}\boldsymbol{x}_t)$. Then compute $\boldsymbol{y}_t = \Theta_{hy}\boldsymbol{h}_t$.

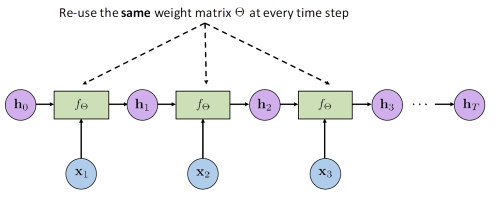
### 1.4.1 Gradient vanishing

RNN with very long sequence suffer from gradient vanishing, where gradients become zeros during backpropagation. ReLU is sometimes problematic.

### 1.4.2 Backpropagation through time (BPTT)

Most common method used to train RNNs.

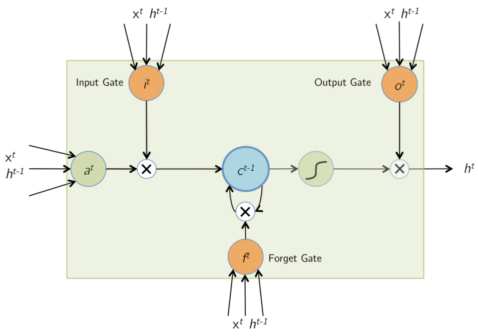
- The unfolded network (used during forward pass) is treated as one big FFN that accepts the whole time series as input
- The weight updates are computed for each copy in the unfolded network, then summed (or averaged) and applied to RNN weights
- In practice, truncated BPTT is used: run the RNN forward $k_1$ time steps, propagate backward for $k_2$ time steps



**BPTT and computation graph**

### 1.4.3 Long short-term memory (LSTM)

Add a memory cell that is not subject to matrix multiplication or squishing (e.g., sigmoid), thereby avoiding gradient decay.



**Gate Computation**

$a^t = \tanh(W_c x^t + U_c h^{t-1}) = \tanh(\hat{a}^t)$

$i^t = \sigma(W_i x^t + U_i h^{t-1}) = \sigma(\hat{i}^t)$

$f^t = \sigma(W_f x^t + U_f h^{t-1}) = \sigma(\hat{f}^t)$

$o^t = \sigma(W_o x^t + U_o h^{t-1}) = \sigma(\hat{o}^t)$

**Memory cell**

$c^t = i^t \odot a^t + f^t \odot c^{t-1}$

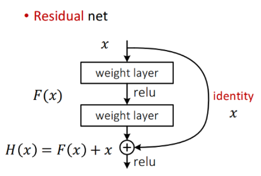
**Output**

$h^t = o^t \odot \tanh(c^t)$

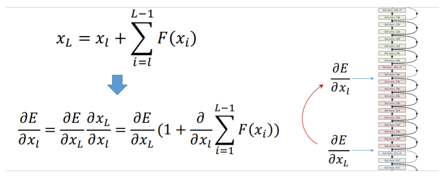
### 1.4.4 Examples

- Many-to-many: machine translation
- Many-to-one: sentence classification
- One-to-many: image captioning

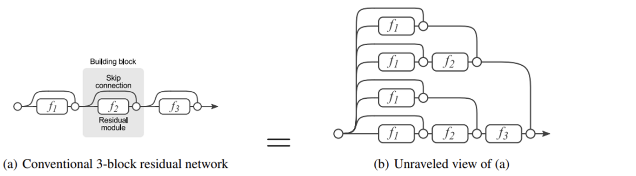
## 1.5 Residual network (ResNet)



(a) ResNet


$x_L = x_l + \sum_{i=l}^{L-1} F(x_i)$
$\frac{\partial E}{\partial x_l} = \frac{\partial E}{\partial x_L}\frac{\partial x_L}{\partial x_l} = \frac{\partial E}{\partial x_L}\left(1 + \frac{\partial}{\partial x_l}\sum_{i=1}^{L-1} F(x_i)\right)$
$\frac{\partial E}{\partial x_l}$
$\frac{\partial E}{\partial x_L}$

(b) This direct path helps maintain the gradient's magnitude and prevents it from vanishing.


Building block
Skip connection
Residual module
$f_1$
$f_2$
$f_3$
(a) Conventional 3-block residual network
=
(b) Unraveled view of (a)

# 2 Ordinary differential equations

## 2.1 Differential equations

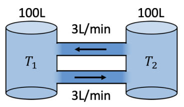
Let $\boldsymbol{h}(t)$ a state vector. $d\boldsymbol{h}(t)/dt$ is a differential equation describing how $\boldsymbol{h}(t)$ change over time. We are interested in solving the following initial value problem (IVP) to know the state in future.

$$\boldsymbol{h}(T) = \boldsymbol{h}(0) + \int_0^T \frac{d\boldsymbol{h}}{dt}dt$$

IVPs are sometimes analytically solved. Otherwise we rely on a solver to approximate the solution.

- $T_1$ has 100 liters of water, and $T_2$ has 100 liters of fertilizer.
- $z(t)=(z_1(t), z_2(t))$ means the amount of fertilizer at time $t$.

$z_1'$ = inflow per minute – outflow per minute = $-0.03\, z_1 + 0.03\, z_2$
$z_2'$ = inflow per minute – outflow per minute = $0.03\, z_1 - 0.03\, z_2$

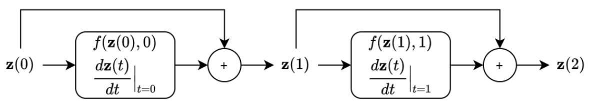


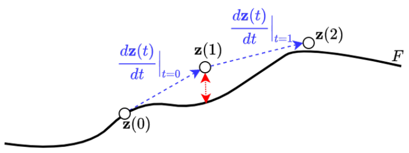
$$\therefore\ z' = Az \text{ or } z' - Az = 0, \text{ where } A = \begin{bmatrix} -0.03 & 0.03 \\ 0.03 & -0.03 \end{bmatrix}$$

- When we have an initial value of $z(0)=(0, 100)$, what is $z(2)$? This kind of problem is called initial value problem (IVP) or forward problem.
- Given data, what is $A$? This kind of problem is called backward problem.

**Example of ODE**

## 2.2 ODE solvers




$\frac{d\mathbf{z}(t)}{dt}\Big|_{t=0}$
$\mathbf{z}(1)$
$\frac{d\mathbf{z}(t)}{dt}\Big|_{t=1}$
$\mathbf{z}(2)$
$F$
$\mathbf{z}(0)$

**Euler method.** Look similar to residual connection

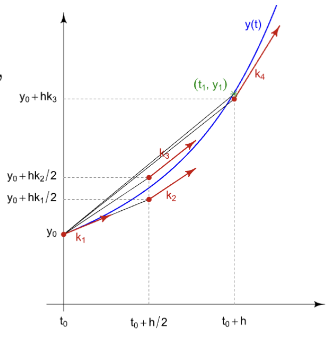
$$\boldsymbol{h}(t+h) = \boldsymbol{h}(0) + hf(\boldsymbol{h}(t)),$$
$$\boldsymbol{h}(t+2h) = \boldsymbol{h}(t+h) + hf(\boldsymbol{h}(t+h)), \quad \cdots$$

Now pick a step-size $h > 0$ and define

$$y_{n+1} = y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$
$$t_{n+1} = t_n + h$$

for $n$ = 0, 1, 2, 3, ..., using[3]

$$k_1 = f(t_n, y_n),$$
$$k_2 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right),$$
$$k_3 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right),$$
$$k_4 = f\left(t_n + h, y_n + hk_3\right).$$



**Runge-Kutta (RK) method**

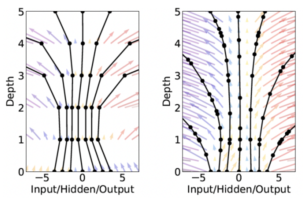
**Dormand–Prince (DOPRI) method**. After comparing the RK4 and RK5 results, use a large step-size $h$ if the difference is small, and small $h$ if the difference is large. In other words, the (adaptive) size-size is inversely proportional to the estimated difference.

## 2.3 Neural ODE

Parameterize the hidden units using ODE specified by neural network:

$$\frac{d\boldsymbol{h}(t)}{dt} = f(\boldsymbol{h}(t), t, \theta)$$

Starting from the input layer $\boldsymbol{h}(0)$, we can define the output layer $\boldsymbol{h}(T)$ to be the solution to this ODE initial value problem at some time $T$. This can be computed by black-box differential equation solver.



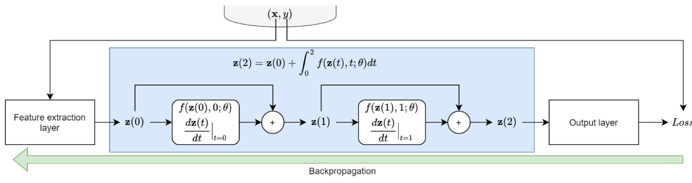
**Left**: ResNet defines a discrete sequence of finite transformations. **Right**: A ODE network defines a vector field, which continuously transforms state. **Circles**: evaluation locations.

Defining/evaluating models using ODE solvers has several benefits:

- **Memory efficiency**: not store intermediate quantities of forward → $O(1)$ memory learning
- Modern ODE solvers provide **error estimate** and evaluation (step size) **adaptive** to given resource

### 2.3.1 NODE-based image classifier

- A typical construction: feature extraction → NODE → output
- NODE layer is analogous to (continuous) residual layers
- Can use standard backpropagation algorithm to train.


$(\mathbf{x}, y)$
$\mathbf{z}(2) = \mathbf{z}(0) + \int_0^2 f(\mathbf{z}(t), t; \theta)dt$
Feature extraction layer
$\mathbf{z}(0)$
$f(\mathbf{z}(0), 0; \theta)$
$\frac{d\mathbf{z}(t)}{dt}\Big|_{t=0}$
+
$\mathbf{z}(1)$
$f(\mathbf{z}(1), 1; \theta)$
$\frac{d\mathbf{z}(t)}{dt}\Big|_{t=1}$
+
$\mathbf{z}(2)$
Output layer
$Loss$
Backpropagation

### 2.3.2 Adjoint sensitivity method

Differentiating through the operations of forward pass is straightforward, but incurs a high memory cost and introduces numerical error. For example, depth of DOPRI frequently becomes large.

We treat the ODE solver as a black box, and compute gradients using the adjoint sensitivity method. Consider optimizing $L()$, whose input is the result of an ODE solver.

$$L(z(t_1)) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta)dt\right) = L(\text{ODESolve}(z(t_0), t_0, T, \theta))$$

We first determine the *adjoint* $\boldsymbol{a}(t) = \partial L/\partial z(t)$. Its dynamics are given by another ODE, which can be thought of as the instantaneous analog of the chain rule:

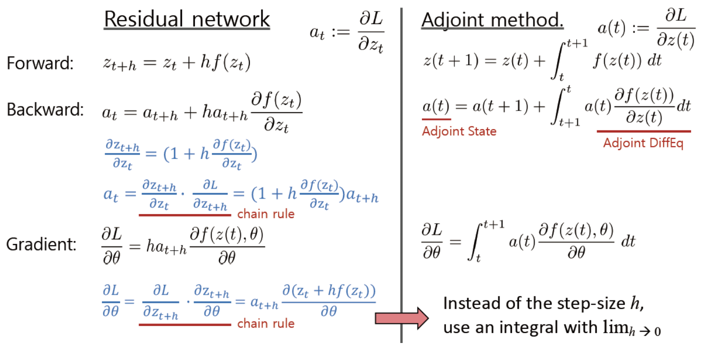
$$\frac{d\boldsymbol{a}(t)}{dt} = -\boldsymbol{a}(t)^T \frac{\partial f(z(t), t, \theta)}{\partial z}$$

We can compute $\partial L/\partial z(t_0)$ by another call to an ODE solver. This solver must run backwards, starting from initial value of $\partial L/\partial z(t_1)$. This will require knowing value of $z(t)$ along its entire trajectory, but we can simply recompute $z(t)$ backwards in time together with the adjoint, starting from its final value $z(t_1)$. We can calculate the gradients with a **reverse-mode integral**,

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \boldsymbol{a}(t)^T \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt$$

No need to maintain computation graph of NODEs → $O(1)$ space.

### 2.3.3 Analogy to ResNet



Residual network $\quad a_t := \frac{\partial L}{\partial z_t}$

Forward: $z_{t+h} = z_t + hf(z_t)$

Backward: $a_t = a_{t+h} + ha_{t+h}\frac{\partial f(z_t)}{\partial z_t}$

$$\frac{\partial z_{t+h}}{\partial z_t} = \left(1 + h\frac{\partial f(z_t)}{\partial z_t}\right)$$

$$a_t = \underbrace{\frac{\partial z_{t+h}}{\partial z_t} \cdot \frac{\partial L}{\partial z_{t+h}}}_{\text{chain rule}} = \left(1 + h\frac{\partial f(z_t)}{\partial z_t}\right)a_{t+h}$$

Gradient: $\frac{\partial L}{\partial \theta} = ha_{t+h}\frac{\partial f(z(t), \theta)}{\partial \theta}$

$$\frac{\partial L}{\partial \theta} = \underbrace{\frac{\partial L}{\partial z_{t+h}} \cdot \frac{\partial z_{t+h}}{\partial \theta}}_{\text{chain rule}} = a_{t+h}\frac{\partial(z_t + hf(z_t))}{\partial \theta}$$

Adjoint method. $\quad a(t) := \frac{\partial L}{\partial z(t)}$

$$z(t+1) = z(t) + \int_t^{t+1} f(z(t))\, dt$$

$$\underbrace{a(t)}_{\text{Adjoint State}} = a(t+1) + \int_{t+1}^{t} \underbrace{a(t)\frac{\partial f(z(t))}{\partial z(t)}dt}_{\text{Adjoint DiffEq}}$$

$$\frac{\partial L}{\partial \theta} = \int_t^{t+1} a(t)\frac{\partial f(z(t), \theta)}{\partial \theta}\, dt$$

Instead of the step-size $h$, use an integral with $\lim_{h \to 0}$

# 3 Transformers

## 3.1 Background

Context vector $c$ is often just $h_T$. Input sequence is bottlenecked through fixed-sized vector. What if sequence is very long?

### 3.1.1 Image captioning with CNN and RNN

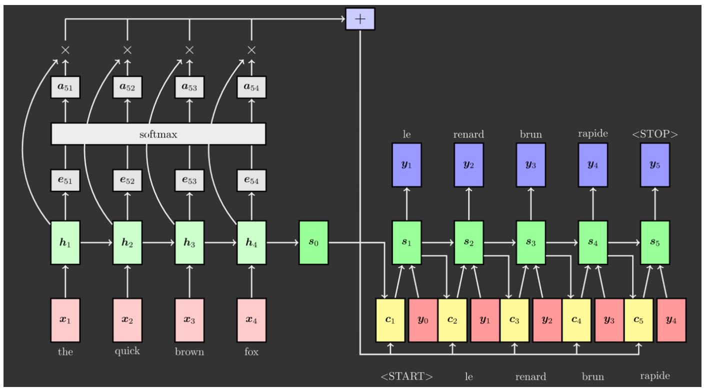
- Transfer learning: take last layer of CNN trained to ImageNet
- Final representation $\boldsymbol{v}$ of CNN is provided to RNN. Now $\boldsymbol{h} = \tanh(W_{xh}\boldsymbol{x} + W_{hh}\boldsymbol{h} + W_{ih}\boldsymbol{v})$
- Sample word and copy to input. Stop after sampling <END> token

## 3.2 Watson-Nadaraya estimator

Data $\{x_1, \cdots, x_m\}$ and ground-truths $\{y_1, \cdots, y_m\}$. Estimate $y$ at a new location $x$. Naive way is just average. Watson-Nadaraya estimator weigh the ground truths: $y = \sum_i a(x, x_i)y_i$ where $x$ is query, $x_i$ is key, and $y_i$ is value.

- Consistency: given enough data, converges to optimal solution
- Simplicity: no free parameters: information is in data not weights
- Deep learning variant: learn the weighting function

### 3.2.1 Seq2Seq with RNN and attention



+
×
$a_{51}$ $a_{52}$ $a_{53}$ $a_{54}$
softmax
$e_{51}$ $e_{52}$ $e_{53}$ $e_{54}$
$h_1$ $h_2$ $h_3$ $h_4$ $s_0$
$x_1$ $x_2$ $x_3$ $x_4$
the quick brown fox
le renard brun rapide <STOP>
$y_1$ $y_2$ $y_3$ $y_4$ $y_5$
$s_1$ $s_2$ $s_3$ $s_4$ $s_5$
$c_1$ $y_0$ $c_2$ $y_1$ $c_3$ $y_2$ $c_4$ $y_3$ $c_5$ $y_4$
<START> le renard brun rapide

RNN need attention for parallelization and deal with long-range dependencies. Here, decoder doesn't use the fact that $\boldsymbol{h}_i$ form an ordered sequence–it just treats them as an unordered set $\{\boldsymbol{h}_i\}$.

## 3.3 Attention

- **Basic attention layer**. Given query $\boldsymbol{q} \in \mathbb{R}^d$, input $\boldsymbol{X} \in \mathbb{R}^{n\times d}$. Similarities $e_i = f_{\text{att}}(q, x_i), \boldsymbol{e} \in \mathbb{R}^n$. Attention weights $\boldsymbol{a} = \text{softmax}(\boldsymbol{e}) \in \mathbb{R}^n$. Output vector $\boldsymbol{y} = \sum a_i \boldsymbol{x}_i \in \mathbb{R}^d$.
- Commonly **separate key and value**: $\boldsymbol{K} = W_k\boldsymbol{X}, \boldsymbol{V} = W_v\boldsymbol{X}$. Similarities $\boldsymbol{E} = f_{\text{att}}(\boldsymbol{K}, \boldsymbol{Q}) \in \mathbb{R}^{n\times n}$, attention weights $\boldsymbol{A} = \text{softmax}(\boldsymbol{E}) \in \mathbb{R}^{n\times n}$. Output vector $y_j = \sum a_{ij}\boldsymbol{v}_i, \boldsymbol{y} \in \mathbb{R}^d$
- **Similarity functions** $f_{\text{att}}$: $q^T x_i$ (dot product), $q^T x_i/\sqrt{d}$ (scaled dot product), $f(Q, X) = Q^T X$ (multiple queries product).
- **Self-attention layer** has one query per input. $\boldsymbol{Q} = \boldsymbol{X} \in \mathbb{R}^{n\times d}$. Uses scaled dot product. Permutation equivariant i.e. $f(s(x)) = s(f(x))$ and works on sets of vectors.

### 3.3.1 Advantages of attention

- Allows decoder to focus on certain parts of source → Significantly improved NMT and time series performance
- Shortcut between faraway states → Mitigates vanishing gradient
- Provides some interpretability

Weighted sum is a selective summary of the information contained in values. Way to **obtain a fixed-size representation** of an arbitrary set of representations (values) dependent on other (query).

Scaled dot-product attention attends to one or few entries in the input key-value pairs. (↔ human) **Multi-head SA** split inputs, use $H$ independent heads in parallel, and concatenate ouputs.

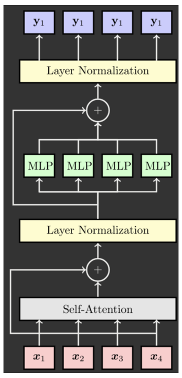
### 3.3.2 Positional encoding

Unlike RNN, attention encoder outputs do not depend on the order of inputs, which is important. Concatenate positional information of input token to input embedding.

$$\text{PE}_{\text{pos},2i} = \sin(\text{pos}/10000^{2i/d}),$$
$$\text{PE}_{\text{pos},2i+1} = \cos(\text{pos}/10000^{2i/d})$$

## 3.4 Transformers

SA is the only interaction between vectors. LayerNorm and MLP work independently per vector. → Highly scalable and parallelizable.



**Transformer block**

# 4 Generative models

A generative task aim to learn $p(x)$ and generate fake samples from learned distribution $p_\theta(x)$.

- Quality vs. diversity dilemma:
  - Likelihood-based GMs directly learns the *pdf* of training data e.g. normalizing flows, VAE.
  - Implicit GMs do not directly maximize the likelihood of training data but implicitly and internally learn it.
- Score-based models (SGMs) propose a novel paradigm i.e., learning the gradient of the log *pdf*, a quantity often known as the (Stein) score function.

## 4.1 Flow-based models

### 4.1.1 Mathematical background

Consider a 2-d coordinate $(x, y)$ and invertible transformation $T$. $(u, v) = T(x, y)$ and $(x, y) = T^{-1}(u, v)$. Jacobian matrix $\mathbf{J}$ is all first-order partial derivatives of this transformation

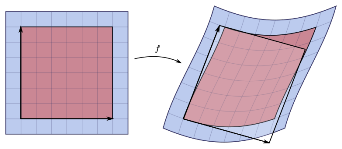
$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \end{bmatrix}$$

Change of Variable theorem (CVT) states

$$\int\int_S f(x, y)\cdot dxdy = \int\int_{T(S)} f(T^{-1}(u, v))\cdot \underbrace{\left|\frac{\partial x}{\partial u}\frac{\partial y}{\partial v} - \frac{\partial x}{\partial v}\frac{\partial y}{\partial u}\right|}_{\text{abs. det. of Jacobian}} dudv$$

CVT for probability density estimation

$$\log p(u, v) = \log p(x, y) + \log\left|\det \frac{\partial T}{\partial(x, y)}\right|$$



**Interpretation of** $dxdy = |\det \partial T/\partial(x, y)|\ dudv$. $f$ sends a small square to a distorted parallelogram. The Jacobian at a point gives the best linear approximation of the distorted parallelogram near that point, and the det $\mathbf{J}$ gives the ratio of the area of the approximating parallelogram to that of the original square.

### 4.1.2 Density estimation in NODEs

Suppose we design a generator using NODEs. $z(0)$ typically follows a unit Gaussian. So we know $\log p(z(0))$. We can estimate $p(z(1))$ as $\log p(z(1)) = \log p(z(0)) + \log|\text{det of Jacobian at} z(0)|$, then $p(z(2))$, and so on. Suppose $z(2)$ is specific image. We know the probability that this specific image generated by the generator.

## 4.2 Generative adversarial networks (GAN)

Zero-sum minimax game between two players

$$\min_G \max_D V(G,D) = \mathbb{E}[\log D(x)]_{x\sim p_{\text{data}}(x)} + \mathbb{E}[\log(1-D(G(x)))]_{z\sim p(z)}$$

To maximize, $D(x)=1$ and $D(G(z))=0$. To minimize, $D(G(z))=1$.

### 4.2.1 Equilibrium state proof of GANs

First consider the optimal discriminator $D$ for any given generator $G$.

**Proposition 1.** *For G fixed, the optimal discriminator D is*

$$D^*_G(\boldsymbol{x}) = \frac{p_{data}(\boldsymbol{x})}{p_{data}(\boldsymbol{x}) + p_g(\boldsymbol{x})} \tag{1}$$

*Proof.* The training criterion for the discriminator D, given any generator $G$, is to maximize the quantity $V(G,D)$

$$\begin{aligned} V(G,D) &= \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x})\log(D(\boldsymbol{x}))dx + \int_z p_z(z)\log(1-D(g(z)))dz \\ &= \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x})\log(D(\boldsymbol{x})) + p_g(\boldsymbol{x})\log(1-D(\boldsymbol{x}))dx \end{aligned} \tag{2}$$

For any $(a,b)\in\mathbb{R}^2\setminus\{0,0\}$, the function $y\to a\log(y)+b\log(1-y)$ achieves its maximum in $[0,1]$ at $a/(a+b)$. The discriminator does not need to be defined outside of $\text{Supp}(p_{\text{data}})\cup\text{Supp}(p_g)$. □

Note that the training objective for $D$ can be interpreted as maximizing the log-likelihood for estimating the conditional probability $P(Y=y|\boldsymbol{x})$, where $Y$ indicates whether $\boldsymbol{x}$ comes from $p_{\text{data}}$ (with $y=1$) or from $p_g$ (with $y=0$). The minimax game is now:

$$\begin{aligned} C(G) &= \max_D V(G,D) \\ &= \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D^*_G(\boldsymbol{x})] + \mathbb{E}_{z\sim p_z}[\log(1-D^*_G(G(z)))] \qquad (3)\\ &= \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D^*_G(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[\log(1-D^*_G(\boldsymbol{x}))] \\ &= \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}\left[\log\frac{p_{\text{data}}(\boldsymbol{x})}{P_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}\right] + \mathbb{E}_{\boldsymbol{x}\sim p_g}\left[\log\frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x})+p_g(\boldsymbol{x})}\right] \end{aligned}$$

**Theorem 1.** *The global minimum of the virtual training criterion $C(G)$ is achieved if and only if $p_g = p_{data}$. At that point, $C(G)$ achieves the value $-\log 4$.*

*Proof.* For $p_g = p_{\text{data}}$, $D^*_G(\boldsymbol{x}) = 1/2$, (consider Eq. 1). Hence, by inspecting Eq. 3 at $D^*_G(\boldsymbol{x}) = 1/2$, we find $C(G) = \log(1/2) + \log(1/2) = -\log 4$. To see that **this is the best possible value** of $C(G)$, reached only for $p_g = p_{\text{data}}$, observe that

$$\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[-\log 2] + \mathbb{E}_{\boldsymbol{x}\sim p_g}[-\log 2] = -\log 4$$

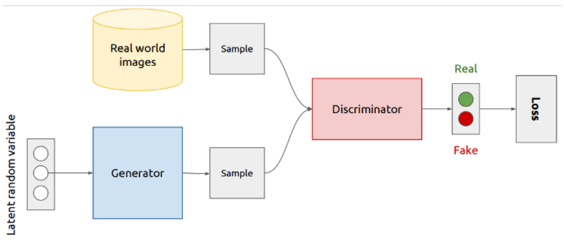
and that by subtracting this expression from $C(G) = V(D^*_G, G)$, we obtain:

$$C(G) = -\log(4) + KL\left(p_{\text{data}}\middle\|\frac{p_{\text{data}}+p_g}{2}\right) + KL\left(p_g\middle\|\frac{p_{\text{data}}+p_g}{2}\right) \tag{4}$$

where KL is the Kullback–Leibler divergence. We recognize in the previous expression the Jensen–Shannon divergence between the model's distribution and the data generating process:

$$C(G) = -\log(4) + 2\cdot JSD\left(p_{\text{data}}\middle\|p_g\right) \tag{5}$$

Since the Jensen–Shannon divergence between two distributions is always non-negative and zero only when they are equal, we have shown that $C^* = -\log(4)$ is the global minimum of $C(G)$ and that the only solution is $p_g = p_{\text{data}}$, i.e., the generative model perfectly replicating the data generating process. □



### 4.2.2 Why Gaussian prior?

Gaussian distributions have the following favorable characteristics:

- The mean of many independent random variables will converge to a Gaussian distribution (*cf.* the central limit theorem)
- Encodes the least amount of prior knowledge i.e., the max entropy) into a model

## 4.3 Score-based generative models

The pdf is defined as $p_\theta(\boldsymbol{x}) = e^{-f_\theta(\boldsymbol{x})}/Z_\theta$. For training, we maximize the log-likelihood $\max_\theta \sum \log p_\theta(\boldsymbol{x})i)$ which is undesirable. We can bypass the intractable $Z_\theta$ by only considering the *score function*, which is the gradient of the log-density *w.r.t.* the random variable.

$$s_\theta(\boldsymbol{x}) = \nabla_{\mathbf{x}}\log p_\theta(\mathbf{x}) = -\nabla_{\mathbf{x}} f_\theta(\mathbf{x}) - \underbrace{\nabla_{\mathbf{x}}\log Z_\theta}_{0} = -\nabla_{\mathbf{x}} f_\theta(\mathbf{x})$$

Score function provides numerical benefits: range $[-\infty,\infty]$ and no need to normalize.

### 4.3.1 Score matching

Given the ground-truth score function, we can minimize the following Fisher divergence.

$$\mathbb{E}_{p(\boldsymbol{x})}[||\nabla_{\mathbf{x}}\log p(\boldsymbol{x}) - s_\theta(\boldsymbol{x})||_2^2]$$

After training, rely on Langevin dynamics to generate samples.

$$\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \epsilon\nabla_{\boldsymbol{x}}\log p(\boldsymbol{x}) + \sqrt{2\epsilon}z_i \quad z_i\sim\mathcal{N}(0,I)$$

**Pitfalls of score matching**: Hard to learn correct score function for low-density regions since the L2 distance is weighted by $p(x)$.

## 4.4 Noise conditional score-based models (NCSMs)

**TL;DR**: Diffuse data with Brownian motion i.e. perturb with Gaussian noise, and reverse this process.

NCSM minimize the following training objective where $i$ is perturbation index:

$$\sum_{i=1}^{L}\lambda(i)\mathbb{E}_{p_{\sigma_i}(\boldsymbol{x})}[||\nabla_{\mathbf{x}}\log p_{\sigma_i}(\boldsymbol{x}) - s_\theta(\boldsymbol{x},i)||_2^2]$$

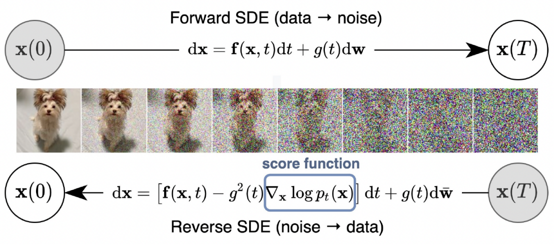
$$\log p_{\sigma_i} = \int \underbrace{p(\boldsymbol{y})}_{\text{GT; hidden}} \quad \underbrace{\mathcal{N}(\boldsymbol{x};\boldsymbol{y},\sigma_i^2 I)dy}_{\text{Gaussian around } y\text{; transition probability}}$$

This is equivalent as learning score function for each $\sigma_i$. NCSM mathematically decompose original loss to constant plus loss of only transition probability, named **denoising score matching loss**.

## 4.5 Score-based generative models

SGMs continuously generalized diffusion models using SDEs.

$$d\boldsymbol{x} = \underbrace{f(\boldsymbol{x},t)dt}_{\text{drift}} + \underbrace{g(t)d\boldsymbol{w}}_{\text{diffusion; } d\boldsymbol{w} \text{ is Gaussian noise}}$$


Forward SDE (data → noise)
$\mathbf{x}(0)$
$\mathrm{d}\mathbf{x} = \mathbf{f}(\mathbf{x},t)\mathrm{d}t + g(t)\mathrm{d}\mathbf{w}$
$\mathbf{x}(T)$
score function
$\mathbf{x}(0)$
$\mathrm{d}\mathbf{x} = [\mathbf{f}(\mathbf{x},t) - g^2(t)\boxed{\nabla_{\mathbf{x}} \log p_t(\mathbf{x})}]\,\mathrm{d}t + g(t)\mathrm{d}\bar{\mathbf{w}}$
$\mathbf{x}(T)$
Reverse SDE (noise → data)

The authors propose 3 types of drift/diffusion: one of them is DDPM, one of them is NCSM.

At $t = T$, original distribution forgotten, noise term dominates; it is single Gaussian. If we know the score function at time $t$ of forward process, we can stochastically reverse that forward process.

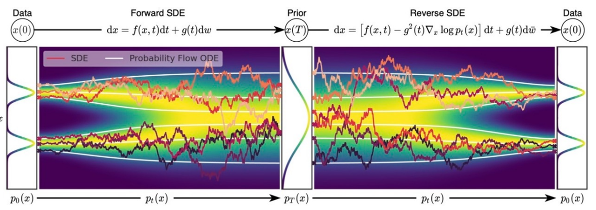
1. Train score network. GT score function replaced to $s_\theta(\boldsymbol{x}, t)$.
   - Also rely on denoising score matching loss
2. Stochastically reverse single Gaussian to target distribution using Euler-Maruyama method

👍 Stochastically reverse → high diversity

👍 High quality (↔ GAN). This minimizes the upper bound of KL. JSD is more stable than KL, but global optimum of GAN is impossible to achieve in practice.

👎 Query score function T times → high complexity

May use probability flow ODE for sampling, which deterministically reverse. Both have the same marginal probability $p_t(\boldsymbol{x})\forall t$.



## 4.6 Poisson flow generative models

### 4.6.1 GNN basics

Degree matrix has number of neighbors at diagonal. **Laplacian** is degree matrix - adjacency matrix. Consider a graph $G$ with Laplacian $L$ and a graph signal (feature) $x \in \mathbb{R}^{N\times D}$ on $G$. Signal $y = Lx$ i.e. $y_i = \sum_{j\in\mathcal{N}_i}(x_i - x_j)$. $y_i$ measures difference between $x$ at a node and its neighborhood, i.e. difference operator.

### 4.6.2 Heat diffusion over graph

Via heat equation $x_t = -Lx$, we say the signal diffuses through graph. $Lx$ is positive if my temperature > neighbor → this equation means 'cooling down'. Temperature at each location is averaged with its neighbors. Converges to average temperature.

The Euler method for temperature update is $x(t+h) = x(t) - hLx(t)$. $h$ can be interpreted as learning rate in DL but step size of Euler method in physics.

### 4.6.3 Graph convolutional network (GCN)

Let $x(t)$ the hidden vector for each node at layer $t$. Consider a row-wise normalized Laplacian $\tilde{L}$. Euler discretized heat equation was $x(t+1) = x(t) - \tilde{L}x(t) = (1 - \tilde{L})x(t)$

GCN uses diffusion process $x(t+1) = \sigma((1-\tilde{L})x(t)W)$. A trainable parameter (or diffusivity/conductivity) $W \in \mathbb{R}^{N\times N}$ represents how close two nodes are. This is just $W$ multiplied to Euler discretized heat equation. The inductive bias of GCN is heat diffusion equation and is influenced by physics (*physics bias*). At the same time, one can consider that this is a first-order graph filtering approach. $\tilde{A}$ is normalized adjacency matrix.

$$\text{Given } \mathrm{H} = \sum_{\ell=0}^{L} \mathrm{h}_\ell \mathrm{S}^\ell, \ell = 1 \text{ and } \mathrm{S} = \mathrm{I} - \tilde{\mathrm{L}} = \tilde{\mathrm{A}}.$$

### 4.6.4 Oversmoothing problem

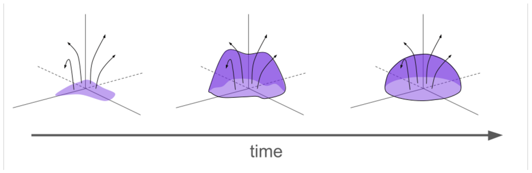
One major problem is oversmoothing problem: all nodes' last **hidden vectors become similar to each other** when the number of GCN layers is large. → mean avg distance (MAD) decreases and node classification accuracy decreases as layer go deeper than some threshold.

Self attention is also GCN, since $1 - \tilde{L} = \tilde{A}$. So transformers also subject to oversmoothing problem. But in case of transformers $\tilde{A}$ is learned not given.

### 4.6.5 Poisson flow models

Idea: at the beginning of sampling, move straight.

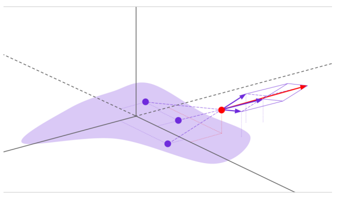
- Diffusion model is inspired from **thermodynamics**: any localized distribution of a gas will eventually spread out to fill an entire space evenly simply through random motion.
- Poisson flow generative models are inspired from **electrostatics**: any distribution of electrons in a hyperplane **generates a uniform hemisphere**.



A charge distribution (purple) and the electric field lines (black) it generates. If we let the charge distribution evolve along the field lines, it will transform into a uniform hemispherical distribution. z-axis corresponds to time of diffusion model.

### 4.6.6 Training Poisson flow models

1. Augment data with $z = 0$
2. Calculate the empirical field in random $x, y, z$–$O(N)$ where $N$ is number of training samples
3. Calculate loss and update **function approximator of empirical field**, $dx = -E(\boldsymbol{x})dt$



### 4.6.7 Sampling Poisson flow models

1. Uniformly sample data on a large hemisphere
2. Use an ODE solver to evolve the points backwards along the Poisson field
3. Evolve backwards until we reach $z = 0$, at which point we have generated novel data from the training distribution.