Deep Learning

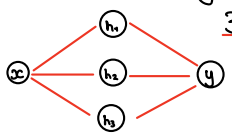
Shallow Neural Networks (Single hidden layer)

Functions $\underline{y} = f[\underline{x}, \phi]$ - map multivariate inputs $\underline{x}$ to multivariate outputs $\underline{y}$

Single Input to Single Output

$$y = f[x, \phi] = \phi_0 + \phi_1 a[\theta_{10} + \theta_{11} x] + \phi_2 a[\theta_{20} + \theta_{21} x] + \phi_3 a[\theta_{30} + \theta_{31} x]$$

Input: x, output: y, Parameters $\phi = \{\phi_0, \phi_1, \phi_2, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$



3 Steps: 1) Compute linear functions:

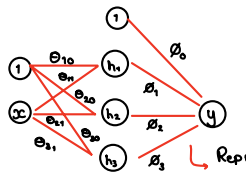
$$z_1 = \theta_{10} + \theta_{11} x, \quad z_2 = \theta_{20} + \theta_{21} x, \quad z_3 = \theta_{30} + \theta_{31} x$$

2) Apply activation functions, a

$$h_1 = a[\theta_{10} + \theta_{11} x] \quad h_2 = a[\theta_{20} + \theta_{21} x] \quad h_3 = a[\theta_{30} + \theta_{31} x]$$

3) Weight three activations and apply offset

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$



↳ Representation with extra node for biases.

Training: Given training dataset $\{x_i, y_i\}_i^I$

Define loss function $L[\phi]$ to measure success/accuracy network.

Optimise (grad descent), $\hat{\phi}$ that minimise $L[\phi]$

a - activation function

$$a(z) = ReLU[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$$

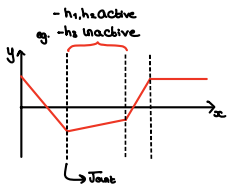
Hidden Units/ Joints

Using the example of Relu, although similiar for other activation functions.

Where three linear function, cross zero (inactive where set to zero).

Each hidden units ⟶ 1 joint

3 hidden units ⟶ 3 joint ⟶ 4 linear regions.



Universal Approximation Theorem

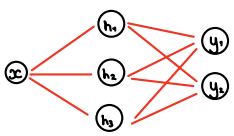
Neural network of D hidden units: D+1 linear regions

$$h_d = a[\theta_{d0} + \theta_{d1} x] \longrightarrow y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

As can simply improve precision by increasing no. neurons.

> There exists a network with one hidden layer containing a finite number of hidden units that can approximate any specificied continuous function on a subset of $\mathbb{R}^n$ to arbitary accuracy.

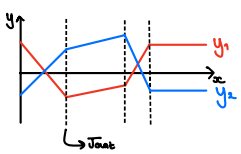
Multivariate Outputs



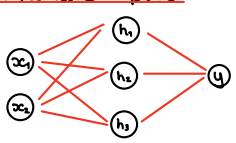
Share the same hidden units/functions:

$$y_1 = \phi_{10} + \phi_{11} h_1 + \phi_{12} h_2 + \phi_{13} h_3$$
$$y_2 = \phi_{20} + \phi_{21} h_1 + \phi_{22} h_2 + \phi_{23} h_3$$

} Thus joints are the same but different linear regions /Scaling/ vertical offset.



Multivariate Inputs:

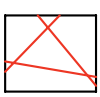


$$h_1 = a[\theta_{10} + \theta_{11} x_1 + \theta_{12} x_2]$$
$$h_2 = a[\theta_{20} + \theta_{21} x_1 + \theta_{22} x_2]$$
$$h_3 = a[\theta_{30} + \theta_{31} x_1 + \theta_{32} x_2]$$

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

2D input - 3D input/output Space

↳ Becomes a plane, Clipped to zero along line



Continuous at boundary - linear planes

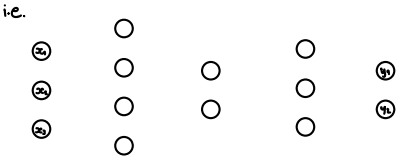
Multi Input - Multi outputs (Single hidden layer)

Share joint/ boundaries. but different Scaling.

Generalised Case: D_o outputs, D_h hidden inputs, D_i inputs:

$$h_d = a\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right] \longrightarrow y_j = \phi_{j0} + \sum_{d=1}^{D_h} \phi_{jd} h_d$$

i.e.



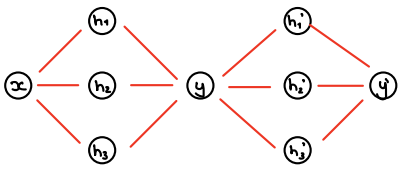
Input, x	Hidden 1, h_1	Hidden 2, h_2	Hidden 3, h_3	Output Layer, y
$D_i = 3$	$D_1 = 4$	$D_2 = 2$	$D_3 = 3$	$D_o = 2$

Terminology

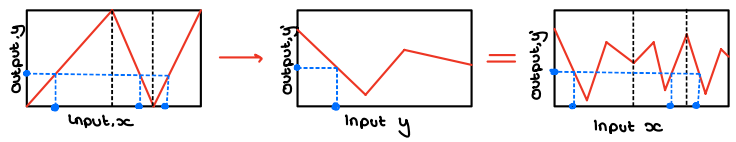
- **Biases** - Y-offsets
- **Weights** - gradients
- **Shallow NN** - 1 hidden layer
- **Deep NN** - 1< hidden layers
- **Fully Connected Network** - all neurons between layers connected
- **Feed forward Network** - no loops
- **Pre-Activation** - Values before activation, $z \rightarrow \theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i$
- **Activations** - Values post activations, $a(z) \rightarrow a\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right]$
- **Capacity** ≈ No. hidden units
- **Depth** ≈ No. hidden Layers
- **Width** ≈ No. Hidden units per layer

Deep Neural Network (more than 1 hidden layer)

Example 1: two Connected Shallow Networks



Consider two independent linear maps:



Network 1: $x \rightarrow y$

$h_1 = a[\theta_{10} + \theta_{11} x]$

$h_2 = a[\theta_{20} + \theta_{21} x]$

$h_3 = a[\theta_{30} + \theta_{31} x]$

$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$

Network 2: $y \rightarrow y'$

$h_1' = a[\theta_{10}' + \theta_{11}' y]$

$h_2' = a[\theta_{20}' + \theta_{21}' y]$

$h_3' = a[\theta_{30}' + \theta_{31}' y]$

$y' = \phi_0' + \phi_1' h_1' + \phi_2' h_2' + \phi_3' h_3'$

Non 1-to-1 mapping:

In the example above: Network 1 → three alternating regions of +ve/-ve Slopes

↳ Different ranges of x mapped to same output $y \in [-1, 1]$

↳ 2nd Mapping: $y \rightarrow y'$ applied three times. → 2nd Network mapping duplicated 3 times.

≈ 9 linear regions. (duplicates/ Stretched / reversed)

$h_1' = a[\theta_{10}' + \theta_{11}' \phi_0 + \theta_{11}' \phi_1 h_1 + \theta_{11}' \phi_2 h_2 + \theta_{11}' \phi_3 h_3]$

or rewritten: $h_1' = a[\psi_{10} + \psi_{11} h_1 + \psi_{12} h_2 + \psi_{13} h_3]$

Similiarly: $h_2' = a[\psi_{20} + \psi_{21} h_1 + \psi_{22} h_2 + \psi_{23} h_3]$

$h_3' = a[\psi_{30} + \psi_{31} h_1 + \psi_{32} h_2 + \psi_{33} h_3]$

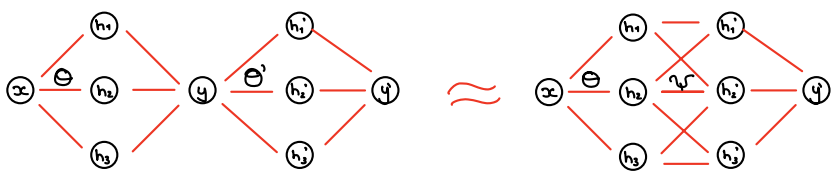
Folding Analogy

Network 1: Created folds in input space x back on itself.

Network 2: Applies function, replicated at all points which were folded on top of each other.

$\psi_{10} = \theta_{10}' + \theta_{11}' \phi_0$, $\psi_{11} = \theta_{11}' \phi_1$, ...

↳ Thus:



Single Input: x

1st Hidden Layer Activations:

$h_1 = a[\theta_{10} + \theta_{11} x]$

$h_2 = a[\theta_{20} + \theta_{21} x]$

$h_3 = a[\theta_{30} + \theta_{31} x]$

2nd Hidden Layer activations

$h_1' = a[\psi_{10} + \psi_{11} h_1 + \psi_{12} h_2 + \psi_{13} h_3]$

$h_2' = a[\psi_{20} + \psi_{21} h_1 + \psi_{22} h_2 + \psi_{23} h_3]$

$h_3' = a[\psi_{30} + \psi_{31} h_1 + \psi_{32} h_2 + \psi_{33} h_3]$

Pre activation - Simply a Shallow Network with 3 outputs.

Post activation, adds new joints to each

Output layer: $y = \phi_0' + \phi_1' h_1' + \phi_2' h_2' + \phi_3' h_3'$

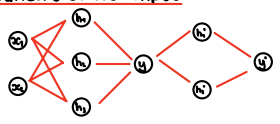
Matrix Notation

1) $\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = a\left[\begin{bmatrix} \theta_{10} \\ \theta_{11} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{11} \\ \theta_{31} \end{bmatrix} x\right]$ $\quad \begin{bmatrix} h_1' \\ h_2' \\ h_3' \end{bmatrix} = a\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{31} & \psi_{32} & \psi_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right]$ $\quad y' = \phi_0' + [\phi_1' \ \phi_2' \ \phi_3'] \begin{bmatrix} h_1' \\ h_2' \\ h_3' \end{bmatrix}$

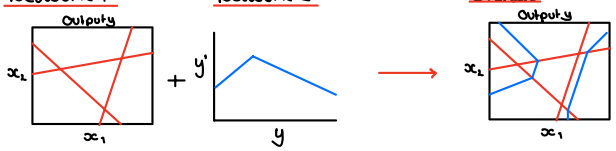
2) $\underline{h} = a[\underline{\theta}_0 + \underline{\theta} x]$ $\quad \underline{h}' = a[\underline{\psi}_0 + \underline{\psi} \underline{h}]$ $\quad y' = \phi_0' + \underline{\phi}' \underline{h}'$

$\underline{h}, \underline{\theta}_0, \underline{h}', \underline{\psi}_0 \rightarrow$ Vectors for each layers activations / biases. $\underline{\psi}$ - Matrix In this example (Single input / output): ϕ_0 - Scalar $\underline{\phi}'$ - vector $\underline{\theta}$ - vector

2 Dimensional Input



Network 1 + Network 2 → Overall



Generalised Notation:

Vector of hidden units at layer k: h_k

Layer k+1: Sum of K-1 Biases, weights and neurons. $h_k = a[\beta_{k-1} + \Omega_{k-1} h_{k-1}]$

For DNN with K layers $y = f[x, \phi]$:

$h_K = a[\beta_{K-1} + \Omega_{K-1} h_{K-1}]$ (last hidden layer)

$y = \beta_K + \Omega_K h_K$ (output layer)

All parameters: $\phi = \{\beta_k, \Omega_k\}_{k=0}^{K}$

Sizes:

β_{k-1}: Vector Size D_k (β_K - Size D_o)

Ω_k: Matrix Size $D_{k+1} \times D_k$ (Ω_0 - Size $D_1 \times D_i$; Ω_K - Size $D_K \times D_o$)

Deep vs Shallow:

For Single Input/Output, ReLu activation

Shallow Network (hidden units D>2)

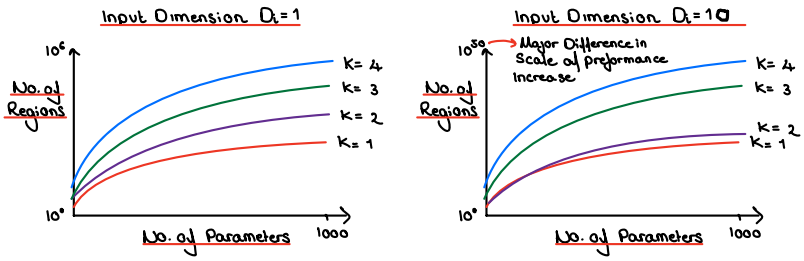
Create $D+1$ linear regions

Using $3D+1$ parameter

Deep Neural Network (K layer of D>2 hidden units)

Create up to $(D+1)^K$ linear regions

Using $3D+1+(K-1)D(D+1)$ parameters



For a fixed parameter budget:

↳ Deep Neural Networks create much more complex functions than Shallow Neural Networks

(this effect is even greater for multi-input dimensions, D_i)

Disadvantages:

↳ DNN contain complex dependencies and symmetries (folding analogy)

So although more regions, may only be advantageous for:

→ Similiar Symmetries in underlying function

→ Underlying function is composition of simpler functions.

Theorem: Maximal no. of linear regions of functions computed by a neural network with n_0 inputs, L hidden layers of width $n_i \geq n_0$ rectifiers at i^{th} layer - lower bounded by:

$$\left(\prod_{i=1}^{L-1} \left[\frac{n_i}{n_0}\right]^{n_0}\right) \sum_{j=0}^{n_0} \binom{n_L}{j}$$

Loss/Cost Functions, $L[\phi]$

Returns a single value which describes the disagreement between the model predictions $f[x_i, \phi]$ and true output y_i

dependent on: $L[\underbrace{\phi}_{\text{Params}}, \underbrace{f[x, \phi]}_{\text{Model}}, \underbrace{\{x_i, y_i\}_{i=1}^{I}}_{\text{Data}}] \rightarrow L[\phi]$

Types of ML Regression

→ Univariate Regression (one real output)

Multivariate Regression (>1 real output)

Classification → Binary Classification - two discrete classes $y \in \{0, 1\}$

Multiclass Classification - >2 discrete classes $y \in \{1, \ldots, K\}$

Probabilistic vs Deterministic Modeling

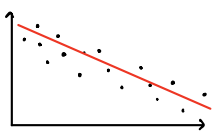
Deterministic Machine Learning

↳ Simply return prediction

↳ (number in regression, relative probability in classification) → Note: not uncertainty in own prediction

No quantification of uncertainty

Prone to overconfidence

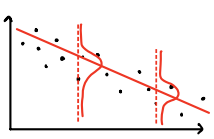


Probabilistic Machine Learning

↳ Model outputs a probability distribution of outputs

↳ Infact parameters which define distribution.

Uncertainty aware / Robust to noise



Probabilistic Machine Learning

Use of Maximum Likelihood

Consider the model as computing a conditional probability for y given input x

Instead of $y_i = f(x_i, \phi) \rightarrow \boxed{\theta_i = f[x_i, \phi] \; + \; y_i \sim P(y|\theta_i)}$

$P(y|x) \rightarrow P(y|\theta) \rightarrow P(y|f[x_i|\phi])$

Loss encourages training data output, y_i to have high probability

Training / Maximum Likelihood Criterion

We choose the model parameters that maximise the combined probability → Assume data is i.i.d – Optimise distributions over Samples.

assuming total likelihood

$$\hat{\phi} = \underset{\phi}{\text{argmax}} \left[\prod_{i=1}^{I} P(y_i|x_i) \right] = \underset{\phi}{\text{argmax}} \left[\prod_{i=1}^{I} P(y_i|\theta_i) \right] = \underset{\phi}{\text{argmax}} \left[\prod_{i=1}^{I} P(y_i|f[x_i|\phi]) \right]$$

↳ Split into 2 parts: i) The underlying probability distribution (θ), $y \sim P(y|\theta)$

ii) How it parameters θ change with x, $\theta = f(x, \phi)$ (machine learning model)

Log-likelihood Criterion

↳ as logarithm monotonically increasing ($z > z'$, $\log[z] > \log[z']$)

best model parameters $\hat{\phi}$: Equal for both $L[\phi]$ and $\log L[\phi]$

$$\hat{\phi} = \underset{\phi}{\text{argmax}} \left[\log \prod_{i=1}^{I} P(y_i|f[x_i, \phi]) \right] = \underset{\phi}{\text{argmax}} \sum_{i=1}^{I} \log \left[P(y_i|f[x_i, \phi]) \right]$$

$$\hat{\phi} = \underset{\phi}{\text{argmin}} \left[\underbrace{-\sum_{i=1}^{I} \log \left[P(y_i|f[x_i, \phi]) \right]}_{L[\phi]} \right]$$

Inference with Probabilistic Machine Learning

↳ As returns distribution over output y rather than $\hat{y}$

Must maximise distribution:

$$\hat{y} = \underset{y}{\text{argmax}} \left[P(y|f[x|f[x, \phi]) \right] \rightarrow$$ often achievable directly from parameters, θ

ie. Normal → Mean, N

Summary of Loss functions

1) Choose underlying probability distribution, $P(y|\theta)$, (of output y given θ)
2) Set ML model to map input → x $\theta = f[x|\phi]$
3) Train model by minimising $L(\phi)$ over training data

$$\hat{\phi} = \underset{\phi}{\text{argmin}} [L[\phi]] = \underset{\phi}{\text{argmin}} \left[-\sum_{i=1}^{I} \log \left[P(y_i|f[x_i, \phi]) \right] \right]$$

4) For test / new data: return distribution $P(y|f[x, \phi])$ or maximum $\hat{y}$

Example 1: Univariate Regression

↳ Single / Scalar output ($y \in \mathbb{R}$) from input

1) Sensible/Example Probability Distribution: eg Normal $P(y|N, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{(y-N)^2}{2\sigma^2} \right]$

2) Set ML model to compute 1 or more parameters $P(y|N, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{(y - f[x, \phi])^2}{2\sigma^2} \right]$

$N = f[x, \phi]$

3) Define Loss function and train by minimising $L[\phi] = -\sum_{i=1}^{I} \log\left[P(y_i|f[x_i,\phi],\sigma^2)\right] = -\sum_{i}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(y_i - f[x_i,\phi])^2}{2\sigma^2}\right]\right]$

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}} -\sum_{i=1}^{I} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{(y_i - f[x_i,\phi])^2}{2\sigma^2} \longrightarrow \boxed{\underset{\phi}{\operatorname{argmin}} \sum (y_i - f[x_i,\phi])^2}$$

↳ ≈ Least Squares for fixed variance.

4) Inference: $\hat{y} = \underset{y}{\operatorname{argmax}}\left[P(y|f[x,\hat{\phi}],\sigma^2)\right]$ For Univariate Normal: $\hat{y} = f[x,\hat{\phi}]$

Note: For variance (fixed), σ^2 (Homeoscedastic)

Can be learned from loss function: $\hat{\phi},\hat{\sigma}^2 = \underset{\phi,\sigma^2}{\operatorname{argmin}} L[\phi]$

Heteroscedastic Regression

Model both Mean and Variance (μ,σ) as function of x.

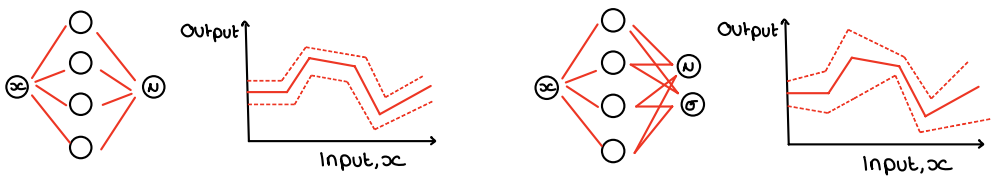
$\mu = f_1[x,\phi]$ $\sigma^2 = f_2[x,\phi]^2$ → (in fact model Std dev and Square to ensure positivity)

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}}\left[-\sum_{i}^{I} \log\left[\frac{1}{\sqrt{2\pi f_2[x_i,\phi]^2}}\right] - \frac{(y_i - f_1[x_i,\phi])^2}{2 f_2[x_i,\phi]^2}\right]$$

→ Optimising ϕ which effect both f_1 and f_2

Univariate (Homeoscedastic) vs Multivariate (Heteroscedastic) Regression

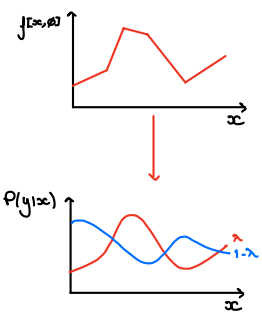
↳ Demonstrated with Shallow Networks



Example 2: Binary Classification

1) Underlying (binary) distribution: $P(y|\lambda) = \begin{cases} 1-\lambda & : y=0 \\ \lambda & : y=1 \end{cases} = (1-\lambda)^{1-y} \cdot \lambda^{y}$

Bernoulli Distribution $y \in \{0,1\}$



2) Set ML model to compute 1 or more parameters

$\lambda \in [0,1]$ $\lambda = \text{Sig}[f[x|\phi]]$

$$P(y|x) = (1 - \text{Sig}[f(x|\phi)])^{1-y} \cdot \text{Sig}(f[x|\phi])^{y}$$

where $\text{Sig}[z] = \frac{1}{1+\exp(-z)}$ maps $[0,1]$

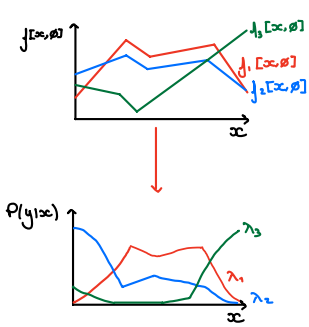
3) Loss function $L[\phi] = -(1-y_i)\log[1-\text{Sig}[f[x_i,\phi]] - y_i \log[\text{Sig}[f[x_i,\phi]]]$ → Binary Cross Entropy

Example 3: Multi class Classification

1) Underlying (binary) distribution: $P(y=k) = \lambda_k$ $\lambda_k \in [0,1]$; $\sum^{K} \lambda_k = 1$

Categorical Distribution:

$y \in \{1,2,\ldots,K\}$



2) ML model: $P(y=k|x) = \text{Softmax}_k[f[x,\phi]]$

Softmax function: $\text{Softmax}_K[z] = \frac{\exp[z_K]}{\sum_{k=1}^{K} \exp[z_K]}$

3) Loss function $L[\phi] = -\sum_{i=1}^{I} \log[\text{Softmax}_{y_i}[f[x_i,\phi]]$ → $\boxed{-\sum_{i=1}^{I} f_{y_i}[x_i,\phi] - \log\left[\sum_{k=1}^{K} \exp[f_k[x_i,\phi]]\right]}$

4) Most Probable Category, $\hat{y}$ $\text{argmax}_k[P(y=k|f[x,\phi])]$

Multiple Outputs

Either - Define a multivariate probability distribution and model with NN

or - Treat each prediction as independent

Treat each output y_d as independent:

(Product of univariate terms for $y_d \in \mathbf{y}$)

$$P(\mathbf{y}|f(\mathbf{x},\phi)) = \prod_d P(y_d|f_d[\mathbf{x},\phi])$$

- d^{th} set of network outputs.
- Parameter distribution y_d

Negative Log Likelihood:

$$L[\phi] = -\sum_i \log(P(\mathbf{y}_i|f[\mathbf{x}_i,\phi])) = -\sum_i \sum_d \log[P(y_{id}|f_d[\mathbf{x}_i,\phi])]$$

- Can use multiple prediction types and treat errors as independent

  ie: two different loss func / distributions Summed.

Cross Entropy Loss

- Equivelent alternative to negative log-likelihood.

Base Idea: Minimise distance between empirical distribution, $q(y)$ of observed data and model distribution $P(y|\theta)$

Measure of Distance:

Kullback-Leibler (KL) divergence

$$D_{KL}[q||p] = \int_{-\infty}^{\infty} q(z)\log[q(z)]\,dz - \int_{-\infty}^{\infty} q(z)\log[p(z)]\,dz$$

Where for $\{y_i\}_{i=1}^I$ (empirical distribution)

$$q(y) = \frac{1}{I}\sum_{i=1}^{I}\delta[y-y_i]$$

δ → Dirac Delta Function

and model distribution $p(z) = P(y|\theta)$

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\left[\int_{-\infty}^{\infty} q(y)\log[q(y)]\,dy - \int_{-\infty}^{\infty} q(y)\log[P(y|\theta)]\,dy\right] \rightarrow \underset{\theta}{\operatorname{argmin}}\left[-\int_{-\infty}^{\infty} q(y)\log[P(y|\theta)]\,dy\right]$$

- $\int q(y)\log[q(y)]\,dy$: No θ dependence
- $-\int q(y)\log[P(y|\theta)]\,dy$: 'Cross entropy' - Amount of uncertainty in one distribution given other

Using def $q(y)$ $\rightarrow$

$$\underset{\theta}{\operatorname{argmin}}\left[-\int_{-\infty}^{\infty}\left(\frac{1}{I}\sum_{i=1}^{I}\delta[y-y_i]\right)\log(P(y|\theta))\,dy\right] \rightarrow \underset{\theta}{\operatorname{argmin}}\frac{1}{I}\left[-\int_{-\infty}^{\infty}\left(\sum_{i=1}^{I}\delta[y-y_i]\right)\log(P(y|\theta))\,dy\right]$$

$$\rightarrow \underset{\theta}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log[P(y_i|\theta)]\right]$$

In probabilistic machine learning: $\theta = f[\mathbf{x}_i,\phi]$, $\theta \rightarrow \phi$

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}}\left[-\sum_i \log(P(\mathbf{y}_i|f[\mathbf{x}_i,\phi]))\right]$$

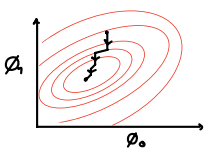
→ Thus shown equivelence to neglog likelihood Criterion.

Optimisation

Optimisation Goal: $\hat{\phi} = \underset{\phi}{\operatorname{argmin}}[L[\phi]]$

The loss of a network is dependent on network parameters

Learn by → Iteratively adjusting ϕ based on gradients of Loss wrt ϕ



Gradient Descent:

Step1: Calculate gradients of loss wrt to parameters.

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix}\frac{\partial L}{\partial \phi_0}\\ \vdots \\ \frac{\partial L}{\partial \phi_n}\end{bmatrix}$$

→ defines 'uphill' direction

Step2: Update parameters

$$\phi_{t+1} = \phi_t - \alpha\cdot\frac{\partial L}{\partial \phi}$$

→ Moves downhill (-ve sign)

α - Learning rate (magnitude of change)

Considering Loss for data set $\{x_i, y_i\}$

↳ $L[\phi] = \sum_i l_i[\phi]$ l_i - individual loss contribution to loss from i^{th} training example

↓

$$\frac{\partial L}{\partial \phi} = \frac{\partial}{\partial \phi} \sum_{i=1}^{I} l_i = \sum_{i}^{I} \frac{\partial l_i}{\partial \phi}$$

Local Minima:

Linear regression

→ Convex loss function - well defined global minimum

Convexity → guarenteed to descent to global minimum.

Generally: Non-Convex

Leads to: Local minima and Saddle points

Alternative: Line Search (Bracketing)

↳ Calculates Gradient less often but Loss more often

1) Calculates negetive gradient (Search direction), d_k
2) Evalates loss at different step sizes along search direction: $x_{k+1} = x_k + \alpha_k d_k$, $f(x_{k+1})$
3) Select step size that minimised function most efficiently.

Stochastic Gradient Descent.

aims to add noise/randomness to gradient at each step

Batches and Epochs

↳ For each iteration, gradient calculated from a random subset (mini-batch)

$$\phi_{t+1} = \phi_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial l_i[\phi_t]}{\partial \phi}$$

↳ Restricted to batch

↳ each subset has its own loss surface

→ Algorithm selects data for batches without replacement.

→ Epoch → Single pass between entire dataset

Properties of SGD

↳ All data points still contribute equally

Less computationally expensive

Can escape local minima/saddle points

Adds noise → Still underlying data so informed/sensible

Learning Rate Schedule

↳ Does not converge as nicely as each batch has unique loss surface/func

Often α decreases for each epoch. → Early: Explore parameter space

Later: Fine tuning

Momentum:

↳ Introduce weighted average of past gradients to update step. (memory of previous gradients) - In SGD retain information from each batch's loss function

$$m_{t+1} = \beta \cdot m_t + (1-\beta) \sum_{i \in \mathcal{B}} \frac{\partial l_i[\phi_t]}{\partial \phi}$$

$$\phi_{t+1} = \phi_t - \alpha\, m_{t+1}$$

→ Accelerates in direction of consistent gradients

Dampens oscillatory behaviour (ie valleys)

Nesterov Accelerated Momentum

Treates momentum term as a prediction of next move.

NAM computes gadients at this predicted point

$$m_{t+1} = \underbrace{\beta \cdot m_t}_{\text{average from previous}} + \underbrace{(1-\beta)}_{\text{correction factor}} \sum_{i \in \mathcal{B}} \frac{\partial l_i[\overbrace{\phi_t - \alpha\beta \cdot m_t}^{\text{jump}}]}{\partial \phi}$$

$\underbrace{\quad\quad}_{\text{gradient at 'hypo' point}}$

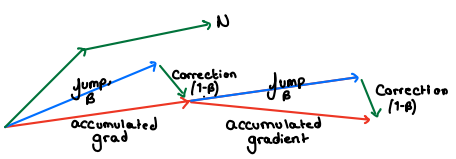
$$\phi_{t+1} = \phi_t - \alpha \cdot m_{t+1}$$

Nesterov Diagram:

↳ Make big jump in direction of previous accumulated gradient ($x_k \rightarrow x_{k+1}$)

Measure gradient in updated position (Look ahead Gradient) (x_{k+1})

Make Correction to overall gradient and use this ($x_k \rightarrow x_{k+1}'$)



Adam

Solved issues with fixed learning rate

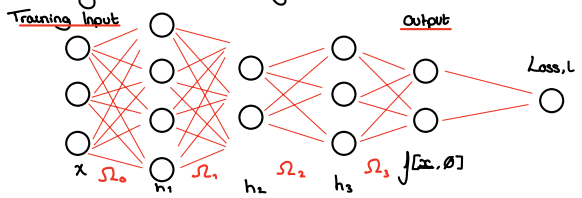
↳ Slow in small gradients, fast in large gradients with no knowledge of curvature. $\left(\frac{\partial L[\phi_t]}{\partial \phi}\right)$ → acts as a proxy for variance using mini batch.

$$m_{t+1} = \beta \cdot m_t + (1-\beta)\nabla L[\phi_t] \qquad \tilde{m}_{t+1} = \frac{m_{t+1}}{1-\beta^{t+1}}$$

$$v_{t+1} = \gamma \cdot v_t + (1-\gamma)(\nabla L[\phi_t])^2 \qquad \tilde{v}_{t+1} = \frac{v_{t+1}}{1-\gamma^{t+1}}$$

$$\phi_{t+1} = \phi_t - \alpha \frac{\tilde{m}_{t+1}}{\sqrt{\tilde{v}_{t+1}} + \varepsilon}$$

Backpropagation: Determining Gradients in Neural Networks:



→ Small changes in Biases and Weights can cause ripple effects throughout network due to compounding/iterative effects.

Base Idea of Back Propagation

Determining how a change to Ω_0 effects loss $(\partial L/\partial \Omega_0)$

↳ How it affects h_1 → How h_1 affects h_2 → How h_2 affects h_3

→ How h_3 affects output → How output affects loss.

back propagation → Can reuse gradient relations working backwards

Backpropagation Algorithm:

For a deep neural network $f[x_i, \phi]$ with inputs x_i, K hidden layers and activation function a. Individual loss function $l_i = l[f[x_i, \phi], y]$

Calculated for every training data point in batch → Summed for update (SGD) step.

Aim: Determine $\frac{\partial l_i}{\partial \beta_k}$ and $\frac{\partial l_i}{\partial \Omega_k}$ for biases and weights, β_k and Ω_k

Forward Pass: Determine preactivations and activations of each layer

$$f_0 = \beta_0 + \Omega_0 x_i$$

$$h_k = a[f_{k-1}] \qquad k \in \{1, 2, \ldots, K\}$$

$$f_k = \beta_k + \Omega_k h_k \qquad k \in \{1, 2, \ldots, K\}$$

Backward Pass:

Gradient Propagation for Whole Layers:

$$\frac{\partial l_i}{\partial f_{k-1}} = a'(f_{k-1}) \odot \Omega_k^T \frac{\partial l_i}{\partial f_k}$$

↳ element-wise multiplication

Ω_k^T can be thought of as $\frac{\partial f_{k+1}}{\partial h_k} = \frac{\partial}{\partial h_k}(\beta_{k+1} + \Omega_{k+1} h_{k+1}) = \Omega_{k+1}^T$

$a'(f_{k-1})$ - Derivative of activation function

Gradient wrt Individual Biases

$$\frac{\partial l_i}{\partial \beta_k} = \frac{\partial l_i}{\partial f_k} \qquad k \in \{K, K-1, \ldots\}$$

Gradient wrt Individual Weights

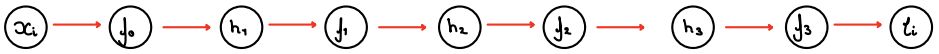
$$\frac{\partial l_i}{\partial \Omega_k} = \frac{\partial l_i}{\partial f_k} h_k^T \qquad k \in \{K, K-1, \ldots\}$$

Exceptions for first layer:

$$\frac{\partial l_i}{\partial \beta_0} = \frac{\partial l_i}{\partial f_0} \qquad \frac{\partial l_i}{\partial \Omega_0} = \frac{\partial l_i}{\partial f_0} \cdot x_i^T$$

In diagrams:

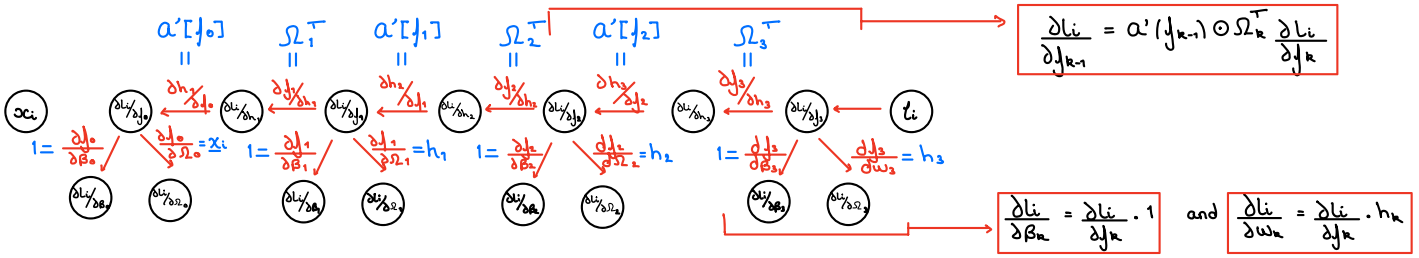
Forward Pass:



$f_0 = \beta_0 + \Omega_0 x_i$ $\quad h_k = a[f_{k-1}]$ $\quad k \in \{1,2,\dots,K\}$

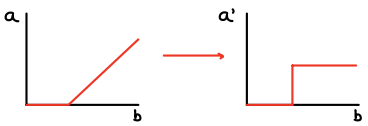
$f_k = \beta_k + \Omega_k h_k$ $\quad k \in \{1,2,\dots,K\}$

Backwards Pass:



Relu activation function - back propogation

$a = \text{ReLU}[b]$ $\quad \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \text{ReLU}(b_1) \\ \text{ReLU}(b_2) \\ \text{ReLU}(b_3) \end{bmatrix}$ $\quad \frac{\partial a}{\partial b} = \begin{bmatrix} \mathbb{I}[b_1>0] & 0 & 0 \\ 0 & \mathbb{I}[b_2>0] & 0 \\ 0 & 0 & \mathbb{I}[b_3>0] \end{bmatrix}$ (ie pointwise multiplication $\odot$ - $\mathbb{I}(b>0)$



Forward vs Reverse Mode

Forward Mode:

Primals →
Derivates →

Propogates Derivates forwards

Each intermediate value differentiated wrt input variables

Complexity: $\sim\Theta(n)$ - Scales with inputs, n. (memory)

Reverse Mode:

Primals →
Derivatives ←

Backwards pass propogates from outputs to inputs

Complexity: $\sim\Theta(m)$ - Scales with outputs, m (memory)

Parameter Initialisation

Forward pass of a general NN:

$f_k = \beta_k + \Omega_k \cdot h_k$
$\quad = \beta_k + \Omega_k \cdot a[f_{k-1}]$

Initialisation: $\beta_k = 0 \ \forall k$

$\Omega_k \sim N(0, \sigma^2)$

Qualitatively:

For very Small Variance, σ^2:

↳ Each layer becomes a weighted sum of h_k - very small due to weights.

Additionally, ReLU clips negative values so ~½'s width of $f_{k-1} \to h_k$

Overall activations Shrink across layers → Vanishing activations

↳ Numerical issues: Too large / Small for finite precision floating point arithmetic

For very Large Variance, σ^2:

↳ With large weights, weighted sum likely very large.

ReLU clips negative values but if σ^2 sufficient large:

Activations grow across layers → Exploding activation

Similiarly for backwards pass

$\frac{\partial l_i}{\partial f_{k-1}} = a'(f_{k-1}) \odot \Omega_k^T \frac{\partial l_i}{\partial f_k}$

Each gradient update - multiplying by Ω^T

Vanishing Activations → Vanishing Gradients → Slow / Inefficient learning - parameter updates.

Exploding Gradients → Exploding Gradients → Uncontrolled / unstable learning

Quantatively:

Considering layer preactivations $f \rightarrow f'$ with Dimensions D_h and $D_{h'}$:

$h = a[f]$

$f' = \beta + \Omega h$

Preactivation in layer f have variance $\sigma_{f_i}^2$

Biases $\beta_i = 0$, weights $\Omega_{ij} \sim N(0, \sigma_\Omega^2)$

Mean + Variance of f'

Mean: $E[f_i']$

$$E[f_i'] = E\left[\beta_i + \sum_{j=1}^{D_h} \Omega_{ij} h_j\right] = E[\beta_i] + \sum_{j=1}^{D_h} E[\Omega_{ij} h_j]$$

$$= \cancel{E[\beta_i]}^{0} + \sum_{j=1}^{D_h} \cancel{E[\Omega_{ij}]}^{0} E[h_j] = 0$$

$\mu_{f_i'} = 0$

$\sigma_{f_i'} = \frac{1}{2} D_h \sigma_\Omega^2 \sigma_f^2$

Variance: $E[f_i'^2] - E[f_i']^2$

$$\sigma_{f_i'}^2 = E[f_i'^2] - E[f_i']^2 = E[f_i'^2] = E\left[\left(\beta_i + \sum_{j=1}^{D_h} \Omega_{ij} h_j\right)^2\right]$$

$$= E\left[\left(\sum_{j=1}^{D_h} \Omega_{ij} h_j\right)^2\right] = \sum_{j=1}^{D_h} E[\Omega_{ij}^2] E[h_j^2]$$

$$= \sigma_\Omega^2 \sum_{j=1}^{D_h} E[h_j^2] = \sigma_\Omega^2 \sum_{j=1}^{D_h} \frac{\sigma_{f_j}^2}{2} = \frac{1}{2} D_h \sigma_\Omega^2 \sigma_f^2$$

↳ Assume f_j are Symmetrically distributed about Zero

∴ ReLU will clip ½ of preactivation - ½ variance

He Initialisation

↳ To ensure variance to be constant. i.e. $\sigma_f^2 = \sigma_{f'}^2$

$$\sigma_\Omega^2 = \frac{2}{D_h}$$

Similiar by back propogation of gradients:

$$\sigma_\Omega^2 = \frac{2}{D_{h'}}$$

For non-Square Ω

↳ $D_h \neq D_{h'}$ - no. hiddenunits in layer differs

↳ Must Compromise ie $\sigma_\Omega^2 = \frac{2}{\left(\frac{D_h + D_{h'}}{2}\right)} = \frac{4}{D_h + D_{h'}}$

Sources of Error

Noise:

Inherent uncertainty in true mapping from input to output.

↳ Result of geniune Stochastic Nature, limited precision of measurement, unobserved/latent variables...

Introduces error for a perfectly fitted mode - unavoidable test error.

Bias:

'Systematic deviation from mean of true function we are modelling - limitations in our model'

↳ result of difference between true function and model Complexity - thus remains in infinite data regime

→ Increase Model Complexity

↳ greater ability to represent true function (greater ability to overfit).

Variance:

Uncertainty in fitted model due to choice of training set

↳ Due to Statistical fluctuations in finite data regime. Model is effected depending on unique subset (overfitting)

→ Reduce by increasing quantity of data

For fixed model (Complexity)

Mathematical Formulation - Mean Square Error

Data generation with noise σ^2:

Expected value $N[x] = E_y[y[x]] = \int y[x] P(y|x)\, dy$ → expectation of observed values.

Noise: $\sigma^2 = E_y[(N(x) - y(x))^2]$

Model prediction: $f[x, \emptyset]$ Observed value: $y[x]$

MSE Loss: $L[x] = (f[x, \emptyset] - y[x])^2$

$= ((f[x, \emptyset] - N(x)) + (N(x) - y[x]))^2$

$= (f[x, \emptyset] - N(x))^2 + 2(f[x, \emptyset] - N(x))(N(x) - y(x)) + (N(x) - y(x))^2$

$\mathbb{E}_y[L(x)] = (f[x, \emptyset] - N(x))^2 + 2(f[x, \emptyset] - N(x))\,\mathbb{E}[(N(x) - y(x))] + \mathbb{E}[(N(x) - y(x))^2]$

$N(x) - N(x) = 0$ Noise: σ^2

$= (f[x, \emptyset] - N(x))^2 + \sigma^2$

↳ $\emptyset$ is a function of training dataset $D \rightarrow \emptyset[D]$

Expectation over datasets (thus $\emptyset$)

$f_N[x] = \mathbb{E}_D[f[x, \emptyset[D]]]$

$(f[x, \emptyset] - N(x))^2$

$= (f[x, \emptyset] - f_N(x) + f_N(x) - N(x))^2$

$= (f[x, \emptyset] - f_N(x))^2 + 2(f[x, \emptyset] - f_N(x))(f_N(x) - N(x)) + (f_N(x) - N(x))^2$

$\mathbb{E}_D[(f[x, \emptyset] - N(x))^2] = \mathbb{E}_D[(f[x, \emptyset] - f_N(x))^2] + 2\,\mathbb{E}[f[x, \emptyset] - f_N(x)]\,(f_N(x) - N(x)) + \mathbb{E}_D[(f_N(x) - N(x))^2]$

Variance of model; $f_N(x) - f_N(x) = 0$; $(f_N(x) - N(x))^2$

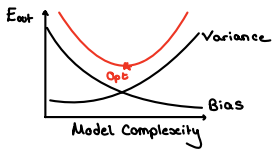
Overall:

$\mathbb{E}_D[\mathbb{E}_y[L[x]]] = \underbrace{\mathbb{E}_D[(f[x, \emptyset[D]] - f_N[x])^2]}_{\text{Variance}} + \underbrace{(f_N[x] - N[x])^2}_{\text{Bias}} + \underbrace{\sigma^2}_{\text{Noise}}$

(Finite Data Model — Infinite Data Model) (Infinite Data Model — True Model)

Bias - Variance Tradeoff:

For finite data regime:



By increasing complexity

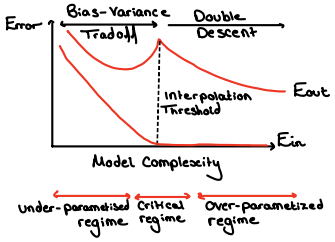
↳ Greater ability to: Represent underlying function → Decrease bias

Overfit to Statistical fluctuations → Increase Variance

Smoother So

Smoother So

Double Descent



Classical ML → Modern ML

Interpolation threshold:

the point where model complexity (no. parameters) is equal to number of training points and thus able to perfectly fit training data (E_{out}) - finite data

Surprise: Test error decreases as model grows into highly overparametised (undersampled regime).

Double descent - the phenomenon where the test error first increases, then decreases, again as model complexity increases

Suggested Reasoning:

At interpolation threshold: model has just enough parameters to perfectly fit noise

↳ Highly oscillatory - poor generalisation - high test error.

↳ lacks ability choose among multiple fits - not able to find smooth, generalisable function

* Regularisation - Restricts parameter space - thus optimises models that perfectly fit

Overparametised Models +

↳ Beyond interpolation threshold - model can perfectly fit data in multiple ways (parameters > threshold)

Implicit Regularisation - biases model towards Simple, Smooth Solutions → generalise well.

↳ favours low norm weights → smoother, more gradual controlled changes.

GD → Favours Low norm Solns } Favour more general

SGD → Favours 'flatter' minima } Solutions.

Regularisation

Restricts parameter space by applying penalties to cost functions which: (reduce generalisation gap between training/test preformances)

→ Reduce overfitting - learning statistical peculiarities

Improve generalisation - for unconstrained areas of no training data.

$$\hat{\phi} = \underset{\phi}{\operatorname{argmax}} \left[\sum_{i=1}^{I} l_i[x_i, y_i] + \lambda \cdot g[\phi] \right]$$

Normal Loss Term | Penalty Loss

λ - hyperparameter of regularisation strength

$g[\phi]$ - large scalar value for less preferrable parameters.

Probabilistic Interpretation

The regularisation term is equivelent to a Prior, $Pr(\phi)$

↳ encompasses 'knowledge/information' on parameters

$$\hat{\phi} = \underset{\phi}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(y_i|x_i, \phi) \right] \longrightarrow \underset{\phi}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(y_i|x_i, \phi) Pr(\phi) \right]$$

Maximium a posteriori - MAP

With $\log[Pr(\phi)] - \lambda \cdot g[\phi]$

L2 - Regularisation (Ridge Regression)

Regularisation term, $g[\phi] = \sum_j \phi_j^2$, L2 Norm

(Typically applied to weights not biases)

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}} \left[\sum_{i=1}^{I} l_i[x_i, y_i] + \lambda \sum \phi_j^2 \right]$$

Encourages Smaller weights

↳ Smoother output function

Explanation:

↳ Adding regularisation - must trade of adherence to data point with desire to be 'smooth'.

In over parameterised regime - extra capacity can adjust regions of no data (interpolation)

↳ encourage this to be smooth.

Implicit Regularisation: Gradient descent

In Infitesimal Step Size (Perfect descent):

$$\frac{d\phi}{dt} = -\frac{\partial L}{\partial \phi}$$

For discrete Step Size, α:

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial L[\theta_t]}{\partial \phi}$$

Deviates from Continuous path.

Equivelent Loss function:

Arrives at same place as discrete

$$L_{GD}[\phi] = L[\phi] + \frac{\alpha}{4} \left\| \frac{\partial L}{\partial \phi} \right\|^2$$

Equivelent Loss function:

[Seek to find correction $g_1[\phi]$ to continuous version S.t gives discrete version]

(use backwards error analysis)

For $\alpha \to 0$ Step Size:

$$\frac{d\phi}{dt} = g[\phi] \longrightarrow \frac{d\phi}{dt} \approx g[\phi] + \alpha g_1[\phi]$$

Taylor Expansion, $\alpha = (\phi - \phi_0)$

$$\phi \approx \phi + \alpha \frac{d\phi}{dt} + \frac{\alpha^2}{2} \frac{d^2\phi}{dt^2}\Big|_{\phi=\phi_0} \longrightarrow \phi + \alpha(g[\phi] + \alpha g_1[\phi]) + \alpha^2/2 \left(\frac{\partial g[\phi]}{\partial \phi} \frac{d\phi}{dt} + \alpha \frac{\partial g_1[\phi]}{\partial \phi} \frac{d\phi}{dt} \right)\Big|_{\phi=\phi_0}$$

$$\longrightarrow \phi + \alpha(g[\phi] + \alpha g_1[\phi]) + \frac{\alpha^2}{2} \left(\frac{\partial g[\phi]}{\partial \phi} g[\phi] + \alpha \frac{\partial g_1[\phi]}{\partial \phi} g[\phi] \right)\Big|_{\phi=\phi_0}$$

↳ discrete trajectory repelled from places where the gradient norm is large (surface is steep)

'changes effective loss surface'

$$\approx \underbrace{\phi + \alpha g[\phi]}_{\phi_0 + \alpha g[\phi_0]} + \alpha^2 \underbrace{\left(g_1[\phi] + \frac{1}{2}\frac{\partial g[\phi]}{\partial \phi} g[\phi] \right)\Big|_{\phi=\phi_0}}_{\text{For equally must} = 0} \quad \text{(remove } \mathcal{O}(\alpha^3)\text{)}$$

Discrete case

$$g_1[\phi] = -\frac{1}{2}\frac{\partial g[\phi]}{\partial \phi} g[\phi] \longrightarrow \frac{\partial \phi}{\partial t} = -\frac{\partial L}{\partial \phi} - \frac{\alpha}{2}\left(\frac{\partial^2 L}{\partial \phi^2}\right)\frac{\partial L}{\partial \phi}$$

$$\longrightarrow \boxed{\tilde{L}_{GD}[\phi] = L[\phi] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \phi}\right\|^2}$$

Implicit Regularisation: Stochastic GD

$$\tilde{L}_{SGD}[\phi] = \underbrace{L[\phi] + \frac{\alpha}{4}\left\|\frac{\partial L}{\partial \phi}\right\|^2}_{\tilde{L}_{GD}[\phi]} + \frac{\alpha}{4B}\sum_{b=1}^{B}\left\|\frac{\partial L_b}{\partial \phi} - \frac{\partial L}{\partial \phi}\right\|^2$$

↳ B - no. batches

L_b = Loss of b^{th} batch

$$L = \frac{1}{I}\sum l_i[x_i, y_i] \qquad L_b = \frac{1}{|\mathcal{B}|}\sum_{i \in b} l_i[x_i, y_i]$$

SGD implicitly favours: Stable gradients (penalises variance between batches)

↳ encourages solutions where all data fits well

- Small batch variance - better generalisation

(vs some data very well and other data less so).

Early Stopping:

Ensembling

Dropout

Applying Noise

Bayesian Inference

Convolutional Networks

Motivation

Fully Connected Neural Networks:

↳ Treat every input as independent (no notion of nearby)

All relationships treated equally – flat unordered vectors

No spatial awareness ⟶ No generalisation across transforms (translation, rotation, etc.)

Each transform must be 'relearnt' ⟶ Redundant Learning ⟶ Inefficient

Convolutional Layers

↳ Leverage shared parameters and localised processing

More generalisable and scaling

Efficiently 'capture' local patterns and spatial relationships.

Why CNN's preform better?

↳ Great inductive bias – interpolates training data better – incorporated prior information – forced to process each input in the same way (data generated by random translation).

Intuition

FCN – Learn every input digit's template

CNN – Share information across positions (Search for smaller range of input/output mappings)

Advantages

Parameter Efficiency:

CNN's reuse a small set of weights (filter/kernel) across input.

'Weight sharing' ⟶ less overall parameters / more memory efficient

Local feature extraction

Data often exhibits local dependencies (ie adjacent pixels / consecutive time steps)

CNN process small regions and scan across data to capture patterns.

Hierarchical Feature Learning:

Stacking multiple Conv. layers allows network to detect low-level features

In early layers ⟶ High level abstractions later on.

↳ Improves generalisation

Translation Invariance

Function/Network, $F(x)$ is invariant on data x with transformation, $t(x)$:

$\boxed{f(t[x]) = f(x)}$ ie output unchanged by transformation

CNN's shared filters/kernels scan across input

↳ detect patterns independently of absolute position

Equivariance for Structured Objects

When a function's, $f[x]$, output is transformed, $t[x]$ by:

$\boxed{f(t[x]) = t(f[x])}$

CNN's encompass this through 'convolution' and shifting of kernel outputs.

Convolutional Layers ⟶ Equivariant to translation

Pooling Mechanisms ⟶ Invariant to translation.
(ie passing to fully connected)

Convolution Operation

1-D case: Input vector, x ⟶ output vector, z s.t:

↳ each output z_i – weighted sum of nearby inputs ⟶ Kernel Size: Size of region kernel takes inputs

Same weights used for all positions ⟶ Convolution Kernel / Filter

Input: $\underline{x} = [x_1, x_2, \dots, x_z]$

For kernel size = 3:

Kernel: $\underline{w} = [w_1, w_2, w_3]^T$

$\boxed{z_i = w_1 x_{i-1} + w_2 x_i + w_3 x_{i+1}}$ ⟶ $i \to i+1$, $z_i \to z_{i+1}$ (equivarient wrt translation

Padding:

Edge Cases: ie first and last inputs ⟶ no previous/subsequent input exists.

Zero-padding: Assumes input is zero outside inputs valid range. (Alternative: Cyclic/reflected weights)

or Valid-Convolutions: discard output positions where kernel exceeds input position range

↳ layers/representation must decrease

Convolution Layers Vocabulary:

Stride, S

↳ the step size with which the kernel moves across the input.

eg $(y * w)[i] = \sum_{k=0}^{K-1} w[k] \cdot x[iS-k]$ → S=1: Kernel moves one step at a time – 1 output per position

S>1: Kernel skips positions/configuration

Kernel Size, K (no. of inputs)

↳ Region of width convolution operation integrates information over. (Width of the receptive field)

Typically odd: Symmetrically centered around the current position

eg $(y * w)[i] = \sum_{k=0}^{K-1} w[k] \cdot x[i-k]$

Dilation rate, D

↳ Introduces gaps between / intersperse kernel elements

↳ Increases receptive field without no. parameters

eg $(y * w)[i] = \sum_{k=0}^{K-1} w[k] \cdot x[i-kD]$

Convolution layer

↳ encompasses i) Convolves input with kernel
ii) Adds bias
iii) Pass through activation function.

Channels (Parallel Kernels on Same Input)

↳ Usually compute several convolutions in parallel → prevent information loss.

Each kernel creates set of hidden variables —Stacked→ feature map or channel

Multi-Channel Systems:

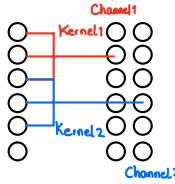
Multiple input channels → Multiple output channels.

Kernels act across multi inputs —Pool→ Single output channel.

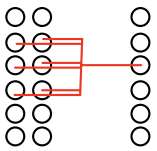
↳ 1 Kernel: N input channels → 1 output channels

N' Kernels: N input channels → N' output channels

Multi output Channel (Multi Kernels)



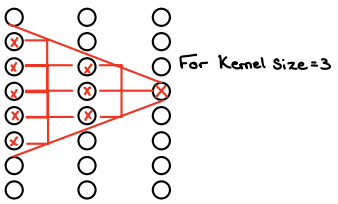
Multi Input Channel



↳ Typically kernels applied to the same input space have equal structure – Simplify dependencies across output channels.

Receptive fields

The receptive field of a neuron is the subset of the input space that influences the neuron's activation.



2D Inputs to Convolutional Networks

↳ Weighted sum over a K×K region (K×K weights)

$$h_{i,j} = a\left[\beta + \sum_{m=1}^{K}\sum_{n=1}^{K} w_{m,n}\, x_{i+m-2,\, j+n-2}\right]$$

w_{mn} - Weight matrix

$h_{i,j}$ – Kernel is translated both horizontally + Verticle

2D output space

Channels in 2D Networks

↳ Multiple 2D outputs (become 3D input × output Space)

RGB Images

↳ 2D input with 3 channels

Kernel ⟶ 3D input matrix • 3D weight matrix.

Each Kernel: Weighted Sum $C_i \times K \times K$ weights + 1 Bias (C_i input Channels)

to Compute C_o output Channels: $C_o \times C_i \times K \times K$ weights + C_o Bias'

DownSampling

↳ Process to reduce Spatial dimensions (width and height) while retaining important features.

Allows Information from large receptive fields to be Combined.

Applied to Channels Independently - preserves number of Channels.

1) Sub-Sampling / Strided Convolution

↳ Effectively retains every S input positions (with Stride = S)

Down Sampling happens Simultaneously to feature extraction

Maintains Sparsity - Does not prioritize important values.

2) Mean/Average Pooling

↳ Seperates Input Space into S×S blocks ⟶ Calculates average value

Retains average representation of local region.

3) Max Pooling

Seperates Input Space into S×S blocks ⟶ Selects maximum value

Preserves Strongest activations

Helps promote translational invariance at small scales.

UpSampling

↳ Increases Spatial resolution (Segmentation, Super-resolution, generative models)

1) Nearest-Neighbour Upsampling

Each pixel value is duplicated S×S times

Results in blocky artifacts / harsh transitio

2) Max unpooling

↳ Undoes Max pooling - places Max in original position and fills rest with zero.

3) Bilinear Interpolation

Estimates missing values by linearly interpolating between between nearest values.

4) Transposed Convolution / Deconvolution

Reverse weight Sharing process - each input contributes a partial Sum of output i.e. transposed Convolution

Adjusting no. Channels (without affecting Spatial resolution)

↳ Apply 1×1 Convolution

For Input Channels C_i ⟶ Output Channels C_o

Weights: $W \in \mathbb{R}^{1 \times 1 \times C_i \times C_o}$

$$y(i,j,c_o) = \sum_{c_i=1}^{C_i} W(c_i, c_o) \cdot x(i,j,c_i) + b(c_o)$$

Applications of CNN

Image classification

Object Detection

↳ Identify and localise multiple objects within an image.

Semantic Segmentation

↳ Assign a label to each pixel in box

Data Augmentation: Method to artificially expand a training dataset by applying transformations to existing data - improve model generalisation by encouraging indifference to transformations

Residual Networks

So far: Sequential Processing

↳ Each layer recieves previous layers output

$h_i = f_i[h_{i-1}, \phi_i]$ - Linear transformation + Activation function

Think of network as series nested functions:

$y = f_3[f_2[f_1[x, \phi_1], \phi_2], \phi_3]$

Limitations of Deep Networks

Although models generally preform better as capacity is added (double descent)

Deep networks - Preformance worsens with more layers

↳ Preformance decreases for both test + training sets.

↳ Not: Inability to generalise functions in deeper networks

but: Ability to train deeper networks. (due to unpredictability of gradients).

Shattered Gradients Problem:

↳ In deep networks, large number of sequential transformations ⟶ gradients fluctuate unpredictably (chaotically)

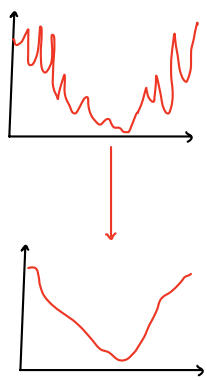
↳ tiny perturbations in input ⟶ Completely change gradient.

Auto corrolation: In deeper networks gradients become uncorrolated and chaotic

Optimization cannot learn stable/meaningful updates.

Changes in early network layers modify the output in increasingly complex ways as the network becomes deeper

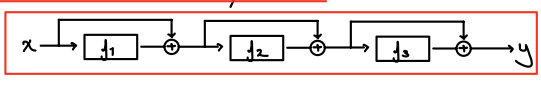
$\frac{\partial y}{\partial f_1} = \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_4}{\partial f_3}$ ⟶ If parameters f_1 change - all derivatives in sequence evaluated at different positions (as dependent on f_1) — updated gradient varies massively



Different failure mode to vanishing/exploding gradients:

↳ Any finite step size — Loss surface may be completely different.

Residual Connections/blocks



$y = h_2 + f_3[h_2, \phi_3] = \underbrace{x + f_1[x] + f_2[x + f_1[x]]}_{h_2} + f_3[\underbrace{x + f_1[x] + f_2[x + f_1[x]]}_{f_3[h_2]}]$

The input to a network layer is added back to the output

Each network layer learns an additive change to the current representation

↳ (rather than full transformation)

Benefits of Residual Networks:

An ensemble of smaller networks whose outputs are summed.

Some paths - pass through fewer layers

Short Paths — Stable gradients — Learn faster (less shattered gradients)

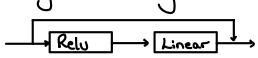
Longer Paths - Capture complex features. (direct path for each layer)

Order of Operation (Relu before linear)

↳ Classically: Relu after linear in each layer

In ResNN: Layers could only increase input

Solution



Initialisation in Residual Networks

Vanishing gradients not an issue → A direct path for each layer

If we use He Initialisation: exponential growth in activation variance

↳ Addition of branches – increases variance $x_{new} = x_{old} + f(x_{old})$ x_{old}: Variance, V x_{new}: Variance, 2V

Solved with:

Batch Normalisation

Shifts/rescales each activation h for given hidden unit, st. its mean + variance are learnable parameters: (Standardise hidden features).

For given batch, B and hidden node:

Mean $\mu_h = \frac{1}{|B|} \sum_{i \in B} h_i$ Variance: $S_h = \sqrt{\frac{1}{|B|} \sum_{i \in B} (h_i - \mu_h)^2}$

Standardise the batch's activations:

$h_i \leftarrow \frac{h_i - \mu_h}{S_h + \varepsilon}$ $\forall i \in B$

Adjust with learnable parameters:

$h_i \leftarrow \gamma h_i + \delta$ $\forall i \in B$

During testing:

do not use test batch Mean/Variance but running average (global distribution) from training

$N_{running} \rightarrow (1-\alpha) N_{run} + \alpha N_B$

$\sigma^2_{run} \rightarrow (1-\alpha) \sigma^2_{run} + \alpha \sigma^2_B$

$\hat{x}_i = \frac{x_i - N_{run}}{\sqrt{\sigma^2_{run} + \varepsilon}}$

CNN batch norm:

↳ Preformed over entire batch size and Kernel outputs (ie values in a given channel)

Layer Norm:

↳ Normalisation across Channels (Kernels) and Spatial positions (of Indiv. Kernel) for individual Samples.

Benefits:

↳ Stable forward propogation

↳ In Res NN: Variance Still increases (additional Source Variance)

Becomes Linear Increase.

(Initially later layers have Smaller effect – tunable δ)

Higher Learning Rates:

↳ Smoother loss Surface and gradients

Regularisation

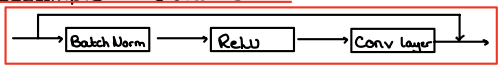
↳ Injects Batch dependent noise (μ_h, S_h)

Improves generalisation.

Initialisation

Initialisation of networks ie Kaiming assume Mean, 0, Variance, 1 to avoid blow out.

Example 'Residual Block'



↳ Residual can be applied (Skip) multiple layers/steps.

U-Map/Hour Glass Networks

↳ Encoder/Decoder

Residual Connections transfer information from encorder → decoder

Transformers

Requirements:

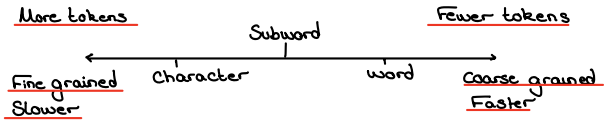
1) Able to efficiently compute large inputs
   - (eg NLP: 100's–1000's words each represented by large encoding vector)
2) Length generalisation – be able to deal with variable input length
3) Attention: attend to previous parts of text for context

Tokenisation

- Each word/subword assigned an index (/token) from vocabulary V of possible tokens
  - Predefined, words/letters/punctuation → (subword tokeniser) → into a token

1) N input tokens → $T \in \mathbb{R}^{|V| \times N}$ – n^{th} column $\approx n^{th}$ token
   - Contains $|V| \times 1$, one hot vector

Trade off in tokenisation:



Embeddings

2) Token matrix mapped to unique (vector) word embeddings in lower-dimensional space (embedding space) – but not 1 hot vector
   - Vocabulary/Embedding Matrix: Ω_e

$$X = \Omega_e T$$

Byte Pair Encoding (BPE) Tokenisation

1) Initialise vocabulary – ie all individual characters in corpus.
2) Count Symbol pairs:
   - Identify most frequent pair of symbols in the corpus.
3) Merge
   - Update vocab with this new unit (subword)
   - Replace all occourences with updated token.
4) Iterate until desired vocab size is reached

Self-Attenuation:

For N inputs $x_1, x_2, \dots x_N$ (of Dimension: $D \times 1$) – each representing a word/token

1) Recomputes word/tokens representation

$$v_m = \beta_v + \Omega_v x_m$$

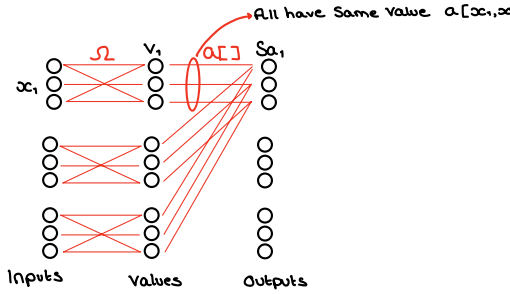
→ β_v – Biases, Ω_v weights

2) For each word n

$$sa_n[x_1, \dots, x_N] = \sum_{m=1}^{N} a[x_m, x_n] v_m$$

→ weighted sum of v_m, weighted by m's relavence to n (n's attention tom)

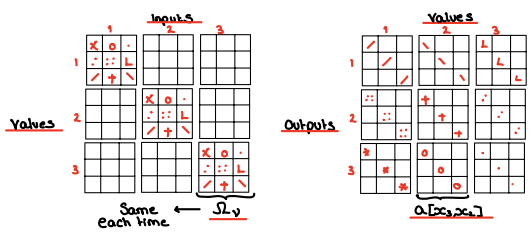
- Scalar weights: $a[x_m, x_n]$ (formerly: $= a[x_m, x_n | \{x_1, \dots x_N\}$)
  - 'attention score': attention n^{th} output pays to x_m (links x_m, x_n)



Attention Weights

- Summed input weights.
- output m

n^{th} weights $a[\bullet, x_m]$ – non negetive and Sum to one
- Each word's relavence to all outputs Sum to Unity



Determing attention weights:

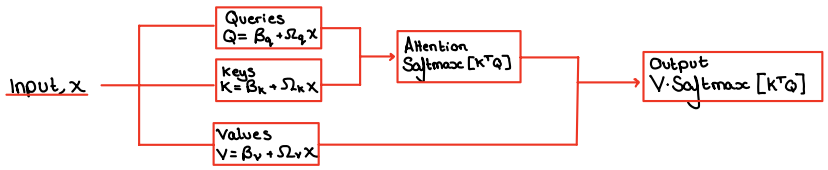
Queries: $q_n = \beta_q + \Omega_q x_n$ – word asking for information

Keys: $K_m = \beta_K + \Omega_K x_n$ – word providing information
- Ω_q, Ω_K – weight matrices

$$a[x_m, x_n] = \text{Softmax}_m[k_\bullet^T q_n] = \frac{\exp[k_m^T q_n]}{\sum_{m'=1}^{N} \exp[k_{m'}^T q_n]}$$

- Fixed Query (Asking Info)
- Sum over all keys (Provided Info)

Overall:



1) $V[x] = \beta_v + \Omega_v x$
2) $Q[x] = \beta_q + \Omega_q x$
3) $K[x] = \beta_k + \Omega_k x$
4) $Sa[x] = V[x] \cdot \text{Softmax}\left[\frac{K[x]^T Q[x]}{\sqrt{D_q}}\right]$

Maintains unit variance

D_q · Dimensionality of vectors

Intuition behind Q, K and V

Queries (Q) - The search criteria for each word in the context of a sentance
↳ (which words are relevant to current word).

Keys (K) - Identifiers that allow words to be matched to queries.
↳ (represent attention a word should recieve from query)

Dot Product (QK^T) - Similarity between each query key pair.
↳ (focus/alignment between queries and keys)

Attention Scores, Attention (Q, K, V): Normalised (probability distribution) of overall importance of each word.
↳ proportion of attention for each word.

Values: Contain actual information about each word.
↳ 'content that will be passed on/carried forward'.

Weighted Sum (Final Output): Contextualised representation of each word.
Incorporates information from all other words in sequence, weighted by relevance.

Scaling

Scaling of dot product Self attention ($1/\sqrt{D_q}$) to improve training:

Without: Dot products of SA can have large magnitudes -
↳ higher dimensionality, greater variance

Largest value may completely dominate - Saturate Softmax ie (0,0,1,0)

Small inputs have little effect - Vanishing gradients → poor training.

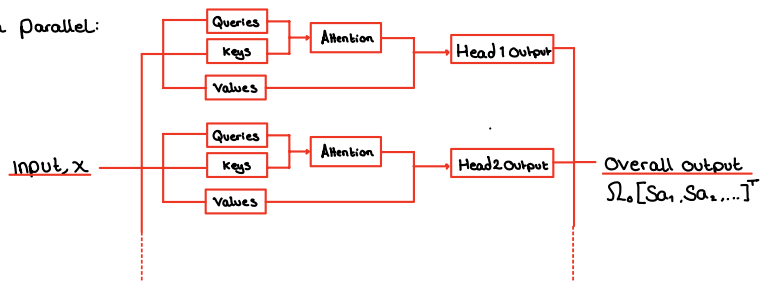
Multiple Heads:

Multi-head Self-attenuation: Multiple mechanisms can be applied in parallel:

h'th SA mechanism/head:

$V_h = \beta_{v,h} + \Omega_{v,h} X$
$Q_h = \beta_{q,h} + \Omega_{q,h} X$
$K_h = \beta_{K,h} + \Omega_{K,h} X$

$\rightarrow Sa_h[x] = V_h \cdot \text{Softmax}\left[\frac{K_h^T Q_h}{\sqrt{D_q}}\right]$



Multi Heads (Global vs Local)

For input dimension: NxD N - no. tokens D - dimension of embedding space.

With H head: Each may focus on subset of embedding space ie Dim (Keys) $\approx D/H \times N$
↳ All samples but different feature space.

"Similiar to CNN's": Each head learns different aspects of relationships between data

Positional Encoding:

The Self attention mechanism loses Sequential Context
↳ does not account for input order x_n

Absolute Positional encoding

↳ Adds a unique positional vector to each token embedding

Matrix π - encodes positional information about input sequence with unique columns.

Ie Input $X' = X + H$

Relative Positional Encodings:

↳ Encodes the relative distance between words directly.

Adds element to attention matrix which encodes particular offset between key position, a and query position, b. - Learn param $\pi_{a,b}$

Ie Attention = Softmax $\left(\frac{Q K^T + \pi_{a,b}}{\sqrt{D}}\right) V$ $\pi_{a,b}$ - relative posn term

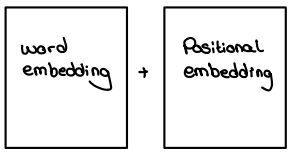
Sinosodial Encoding

$\sin\left(2\pi \frac{n}{\lambda_d}\right)$ n- position index of token

d- wavelength for each embedding dim

Varying frequency - allows capture of both fine (local) and global context

Add Sinosodial information allows model to understand distances between tokens.



Transformer layer

Self Attention - one part of overall transformer layer (multi-head)

↳ allow word representation to interact with one another.

Other features:

→ Residual connections → Layer Norm (normalised across features).

→ Fully Connected Network

Transformer Model:

Encoder: transforms the text embeddings into a representation that can support a variety of tasks.

Decoder: predicts the next token to continue the input text.

Encoder-Decoder: used in sequence-to-sequence tasks: one text string is converted to another.

Encoder (BERT)

Transfer Learning:

1) Pre-training on large text 'corpus' → general language understanding

2) Fine tuning on smaller dataset → Specific NLP tasks (Sentimental analysis, etc.)

Pre training Phase (Self Supervision)

Non manually labelled data.

Improves: Syntax understanding and common sense analysis.

Masked language modeling (MLM)

↳ Random words replaced with <mask> - predicts word.

Next Sentance Prediction (NSP)

↳ Predicts whether two given sentences follow each other.

Fine-Tuning Phase:

Add an output layer and train on task specific data.

Text Classification (Sentiment analysis):

↳ Passed through logistic Sigmoid layer

Word Classification (Named Entity Recognition - NER)

↳ Classifies each word into catagories ie (person, organisation, location)

Passed through Softmax function (multi class classification).

Decoder (GPT3)

↳ Role to generate the next token in a sequence

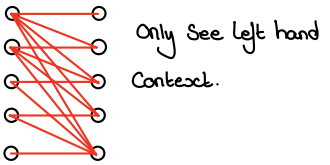
Masked Self Attention:

To only allow context (beforehand) to the masked item.

Attention mechanism by setting future context scores to $-\infty$ before applying Softmax

$q_i \cdot k_{j>i} = -\inf$ (keys to RHS of query.)

↳ Softmax : $\exp(q,k) = 0$.



Steps)

Input ⟶ Embedding ⟶ Masked Self Attention

⟶ Fully Connected Network ⟶ Softmax to all vocab prob.

Loss) Maximise log probabilities of correct next token (cross-entropy loss).

Autoregressive Generation

Once trained, input ⟶ probability distribution of next token from vocab.

Most likely token selected / sampled.

Newly generated token (sequence) fed back through the model.

Variational Autoencoders

↳ Probabilistic generative models – aim to learn distribution $Pr(x)$ over the data.

VAE – Neural architecture designed to help learn

Final Model $Pr(x)$ – Non linear latent variable model

Latent Variable Models:

↳ Indirect approach to describe Multi-Dimensional Probability Distribution $Pr(x)$.

$Pr(x,z)$: joint distribution of data, x and latent variable, z.

Recover $Pr(x)$ from marginalisation of z:

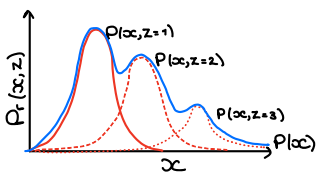
$$Pr(x) = \int Pr(x,z)\, dz. \longrightarrow Pr(x) = \int Pr(x|z)\, P(z)\, dz$$

Typically conditional representation

$$Pr(x,z) = P(x|z)\, P(z)$$

Likelihood: $P(x|z)$ ⟶ Relatively simple $P(x|z)$, $P(z)$

Prior: $P(z)$ Can form complex $P(x)$.



Nonlinear latent variable models:

↳ Both data, x and latent variables are continuous and multivariate.

Prior, $Pr(z)$: Standard multivariate normal.

$$Pr(z) = Norm_z[0, I]$$

Likelihood, $Pr(x|z,\phi)$: Normally distributed.

$$Pr(x|z,\phi) = Norm_x[f[z,\phi], \sigma^2 I]$$

Mean. Non linear function of latent variable

$f[z,\phi]$ → Described by Deep Network, parameters ϕ.

Covariance: Spherical, $\sigma^2 I$.

$$Pr(x|\phi) = \int Pr(x,z|\phi)\,dz = \int Pr(x|z,\phi)\,Pr(z)\,dz$$
$$= \int Norm_x[f[z,\phi], \sigma^2 I] \cdot Norm_z[0, I]\,dz.$$

Summary: Infinite weighted sum of spherical gaussians.

Weights: $Pr(z)$

Means: $f[z, \phi]$

Generation from latent variable model (ancestral sampling):

1) Draw z^* from prior : $z^* \sim Pr(z)$
2) Pass through network for likelihood mean: $f[z^*, \phi]$
3) Draw x^* from likelihood $x^* \sim Pr(x|z^*, \phi)$

Training:

Maximise loglikelihood over training dataset $\{x_i\}_{i=1}^I$ wrt model parameters:

Assuming variance term is known:

$$\hat{\phi} = \underset{\phi}{\text{argmax}} \left[\sum_{i=1}^{I} \log \left[Pr(x_i|\phi) \right] \right]$$

$$Pr(x_i, \phi) = \int Norm_x[f[z,\phi], \sigma^2 I] \cdot Norm_z[0, I]\,dz$$

↳ Intractable: No closed form / easy evaluation.

Evidence Lower Bound (ELBO) on log-likelihood:

Used as proxy for likelihood - depends on additional parameters θ.

Theory: Jensen's Inequality

For a concave function $g[\cdot]$ of the expectation of data y is greater than or equal to expectation of function of data:

$$g[\mathbb{E}[y]] \geq \mathbb{E}[g[y]]$$

Taking Concave function: $g = \log$

$$\log[\mathbb{E}[y]] \geq \mathbb{E}[\log[y]]$$
$$\log\left[\int Pr(y)\,y\,dy\right] \geq \int Pr(y)\log[y]\,dy$$

In fact: $\log\left[\int Pr(y)\,h(y)\,dy\right] \geq \int Pr(y)\log[h[y]]\,dy$ as h simply its own random variable.

Using Jensen's Inequality for ELBO

$$\log Pr(x|\phi) = \log\left[\int Pr(x,z|\phi)\,dz\right]$$

↓ Introduce arbitary probability distribution $q_l(z)$.

$$= \log\left[\int q_l(z)\,\frac{Pr(x,z|\phi)}{q_l(z)}\,dz\right] \geq \underbrace{\int q_l(z)\log\left[\frac{Pr(x,z|\phi)}{q_l(z)}\right]dz}_{ELB} \quad \text{Jensens Inequality}$$

$$ELBO[\theta,\phi] = \int q_l(z|\theta)\log\left[\frac{Pr(x,z|\phi)}{q_l(z|\theta)}\right]dz$$

Introduces θ - Parameters of $q_l(z)$ distribution.

Learning: Must maximise ELBO as both a function of ϕ and θ. → Adjusting θ - Lower bound distance from log likelihood. ↕

Adjusting ϕ - Move along ELBO ↔

Tightness of Elbo:

For a fixed, ϕ: Elbo is tight when ELBO = log likelihood. i.e. optimal $q_\phi(z|\theta)$.

$$ELBO[\theta,\phi] = \int q_\phi(z|\theta)\left[\frac{Pr(x,z|\phi)}{q_\phi(z|\theta)}\right]dz = \int q_\phi(z|\theta)\log\left[\frac{Pr(z|x,\phi)\,Pr(x|\phi)}{q_\phi(z|\theta)}\right]dz$$

$$= \underbrace{\int q_\phi(z|\theta)\log[Pr(x|\phi)]\,dz}_{\text{no } z \text{ dependence}} + \underbrace{\int q_\phi(z|\theta)\log\left[\frac{Pr(z|x,\phi)}{q_\phi(z|\theta)}\right]dz}_{\text{KL Divergence}}$$

$$= \log Pr(x|\phi)\,dz - D_{KL}\left[q_\phi(z|\theta)\,||\,Pr(z|x,\phi)\right]$$

KL Divergence - measures 'distance' between distributions

i.e: $q_\phi(z|\theta) = Pr(z|x,\phi) \longrightarrow$ KL Divergence $= 0$

↳ posterior distribution over LV, z

Alternative expression

$$ELBO[\theta,\phi] = \int q_\phi(z|\theta)\log\left[\frac{Pr(x|z,\phi)\,Pr(z)}{q_\phi(z|\theta)}\right] = \int q_\phi(z|\theta)\log[Pr(x|z,\phi)]\,dz + \int q_\phi(z|\theta)\log\left[\frac{Pr(z)}{q_\phi(z|\theta)}\right]dz.$$

$$ELBO[\theta,\phi] = \int q_\phi(z|\theta)\log\left[Pr(x|z,\phi)\right]dz - D_{KL}\left[q_\phi(z|\theta)\,||\,Pr(z)\right]$$

Variational Approximation:

Assumption: $q_\phi(z|\theta) \sim$ Multivariate Normal Dist - mean, μ, Covariance, Σ

Ideal choice for $q_\phi(z|\theta)$ was posterior $p(z|x,\phi)$ which is data dependent. (by KL Div)

To do same:

$$q_\phi(z|x,\theta) = Norm_z\left[g_\mu[x,\theta], g_\Sigma[x,\theta]\right]$$

- $g[x,\theta]$ is neural network to predict μ, Σ.

Variational Autoencoder:

Build a network that computes ELBO:

$$ELBO[\theta,\phi] = \int q_\phi(z|x,\theta)\log\left[Pr(x|z,\phi)\right]dz - D_{KL}\left[q_\phi(z,x|\theta)\,||\,Pr(z)\right]$$

Where $q_\phi(z|x,\theta) = Norm_z\left[g_\mu[x,\theta], g_\Sigma[x,\theta]\right]$

$$E[\log(x|z,\phi)] \approx \tfrac{1}{N}\sum_n a[z_n^*] \approx a[z^*]$$

↳ Estimated from Single Sample

$$ELBO[\theta,\phi] = \log\left[Pr(x|z^*,\phi)\right] - D_{KL}\left[q_\phi(z,x|\theta)\,||\,Pr(z)\right]$$

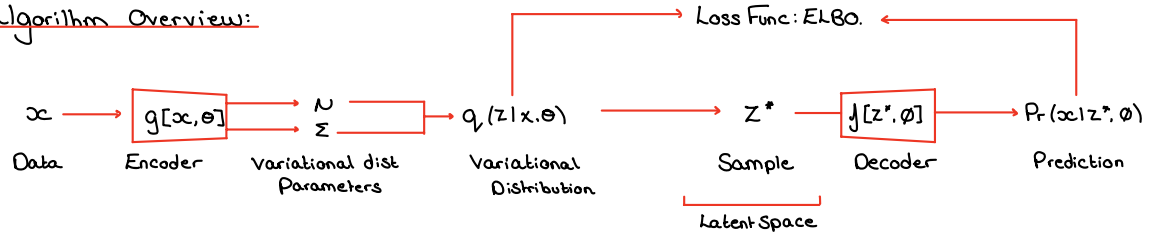
$q_\phi(z|x,\theta) = Norm_z[\mu,\Sigma]$

$Pr(z) = Norm_z[0,I]$

$$D_{KL} = \tfrac{1}{2}\,Tr[\Sigma] + \mu^T\mu - D_z - \log[\det[\Sigma]]$$

D_z - Dimensionality of latent space.

Algorithm Overview:



z - Lower dimensional latent vector

Encoder, $g[x,\theta]$

↳ Mapping from data to latent variable (Parameter of Prob)

(actual z - Sample from latent distribution $q(z|x,\theta)$

Decoder $f[z,\phi]$

↳ Mapping from latent variable to data.

Diffusion Models

An encoder-decoder generative model that learns to generate data by reversing a gradual noising process.

Encoder:

→ Takes input $\underline{x}$ and maps it sequentially through a series of latent variables, $\underline{z}_1, \ldots, \underline{z}_T$ by gradually adding noise to the image (in a prespecified way).

→ With sufficient steps in this markov chain mapping, the conditional $q(z_T|\underline{x})$ and marginal distributions $q(z_T)$ become standard normal distributions.

Decoder

→ A series of networks which aim to map backwards from each pair of latent variables $\underline{z}_t \rightarrow \underline{z}_{t-1}$ encouraging the model to denoise the image and invert the corrosponding encoding step.

Mathematical Formulation

Encoder

$$\underline{z}_1 = \sqrt{1-\beta_1}\,\underline{x} + \sqrt{\beta_1}\,\underline{\varepsilon}_1 \qquad \beta_t \in [0,1] \quad \text{'noise scheduler'}$$

$$\underline{z}_t = \sqrt{1-\beta_t}\,\underline{z}_{t-1} + \sqrt{\beta_t}\,\underline{\varepsilon}_t \qquad \varepsilon_t \sim \text{Std normal noise}$$

$$q(z_1|x) = \text{Norm}_{z_1}\left[\sqrt{1-\beta_1}\,\underline{x}, \beta_1 I\right]$$
$$q(z_t|z_{t-1}) = \text{Norm}_{z_t}\left[\sqrt{1-\beta_t}\,\underline{z}_{t-1}, \beta_t I\right]$$

$$q(z_T \ldots z_1|x) = q(z_1|x)\prod_{t=2}^{T} q(z_t|z_{t-1})$$

Diffusion Kernel

Due to nested model, a closed form mapping is defined.

$$z_t = \sqrt{\alpha_t} + \sqrt{1-\alpha_t}\,\varepsilon \qquad \alpha_t = \prod_{s}^{t}(1-\beta_s)$$
$$q(z_t|x) = \text{Norm}_{z_1}\left[\sqrt{\alpha_t}\,x, (1-\alpha_t)I\right]$$

Problems:

Marginal Distribution:

$$q(z_t) = \int q(z_t|x)\,\underbrace{p(x)}_{\text{unknown}}\,dx$$

Reverse Conditions:

$$q(z_{t-1}|z_t) = \frac{q(z_t|z_{t-1})\,q(z_{t-1})}{q(z_t)}$$

But if x is unknown:

$$\begin{aligned} q(z_{t-1}|z_t) &\propto q(z_t|z_{t-1}, x)\,q(z_{t-1}|x) \\ &\propto q(z_t|z_{t-1})\,q(z_{t-1}|x) \\ &\propto \text{Norm}_{z_t}\left[\sqrt{1-\beta_t}\cdot z_{t-1},\ \beta_t I\right] \cdot \text{Norm}_{z_{t-1}}\left[\sqrt{\alpha_t}\cdot x_1,\ (1-\alpha_t)I\right] \end{aligned}$$

Decoder model

Aim to learn the probabilistic mapping from $z_t \rightarrow z_{t-1}$ for each latent variable.

True reverse distributions $q(z_{t-1}|z_t)$ are complex and dependent on $p(x)$ but we approx as a gaussian → appropriate when β_t close to 0

$$P_r(z_t) = \text{Norm}_{z_T}[0, I]$$
$$P_r(z_{t-1} \mid z_t, \phi_t) = \text{Norm}_{z_{t-1}}[f_t[z_t, \phi_t], \sigma_t^2 I]$$
$$P_r(x \mid z_1, \phi_1) = \text{Norm}_x[f_1[z_1, \phi_1], \sigma_1^2 I]$$

We use a series of neural networks $f_t[z_t, \phi_t]$ to learn the mean of the distribution mappings.

Training a Diffusion Model:

Joint Distribution:

$$P(x, z_1, \ldots z_T \mid \phi_1, \ldots, \phi_T) = P_r(x \mid z_1, \phi_1) \prod_{t=2}^{T} P_r(z_{t-1} \mid z_t, \phi_t) \cdot P(z_t)$$

Marginal Distribution

$$P(x \mid \phi_1, \ldots, \phi_T) = \int P(x, z_1, \ldots z_T \mid \phi_1, \ldots, \phi_T) \, dz_{1\ldots T}.$$

Ideal Goal:

$$\hat{\phi}_{1,\ldots,T} = \text{argmax}_{\phi}\left[\sum_{i=1}^{I} \log\underbrace{[P_r(x \mid \phi_1, \ldots, \phi_T)]}_{\text{True 'Evidence'}}\right]$$

Intractable integrals so formulate using Jensens Inequality and maximise the evidence lower bound using $\{\phi_i\}$ to both produce tightness and maximisation.

$$\log P_r(x \mid \phi_1, \ldots, \phi_T) = \log \int q_\phi(z_1, \ldots, z_T \mid x) \frac{P_r(x, z_1, \ldots, z_T \mid \phi_1, \ldots, \phi_T)}{q_\phi(z_1 \ldots z_T \mid x)}$$

$$ELBO[\phi] = \int q_\phi(z_1, \ldots, z_T \mid x) \log\left[\frac{P_r(x, z_1, \ldots, z_T \mid \phi_1 \ldots \phi_T)}{q_\phi(z_1, \ldots, z_T \mid x)}\right] dz_{1\ldots T}$$

Using Markov Chain Properties:

$$ELBO[\phi] = E\left[\log P_r(x \mid z_1, \phi_1)\right] - \sum_{t=2}^{T} E_{q(z_t \mid x)}\left[D_{KL}\left[q_\phi(z_{t-1} \mid z_t, x) \,\|\, P_r(z_{t-1})\right]\right]$$

We aim to minimise distance between reverse conditional of encoder $P_r(z_{t-1} \mid z_t, \phi_t)$ and true encoder $q_\phi(z_{t-1} \mid z_t, x)$.

Reparametisation:

Rather than $\hat{z}_{t-1} = f_t[z_t, \phi_t]$ modeling the mean output Consider the models $\hat{\epsilon} = g_t[z_t, \phi_t]$ to learn the noise added to x to achieve z_t

$$f_t[z_t, \phi_t] = \frac{1}{\sqrt{1-\beta_t}} z_t - \frac{\beta_t}{\sqrt{1-\beta_t}\sqrt{1-\alpha_t}} g_t[z_t, \phi_t]$$

Becomes a MSE loss term - Summed over all layers:

Loss

$$L[\phi_{1\ldots T}] = \sum_{i}^{I} \sum_{t}^{T} \| g_t[z_{it}, \phi_t] - \epsilon_{it} \|^2$$
$$= \sum_{i}^{I} \sum_{t}^{T} \| g_t[\sqrt{\alpha_t} x + \sqrt{1-\alpha_t}\, \epsilon_{it}, \phi_t] - \epsilon_{it} \|^2$$

In practise - Training

$L[\phi_t]$ we take random data and only train on model t:

g_t by random sample:

For $i \in B$:

$t \sim \text{Uniform}[1, \ldots, T]$

$\varepsilon \sim \text{Norm}[0, I]$

$l_i = \| g_t[\sqrt{\alpha_t} x_i + \sqrt{1-\alpha_t}\,\varepsilon, \phi_t] - \varepsilon \|^2$

Generation:

Although we train $\hat{\varepsilon} = g_t[z_t, \phi_t]$ to learn the mapping from $x \rightarrow z_t$ we rescale these and apply these at every step backwards to improve accuracy.

Note $f_{z_{t-1}}[z_t, \phi_t]$ aimed to learn the mean so additional noise is applied:

$z_T \sim \text{Norm}_z[0, I]$

$$\hat{z}_{t-1} = \frac{1}{\sqrt{1-\beta_t}} z_t - \frac{\beta_t}{\sqrt{1-\alpha_t}\sqrt{1-\beta_t}} g_t[z_t, \phi_t]$$

$$z_{t-1} \sim \text{Norm}[\hat{z}_{t-1}, \sigma_t \varepsilon]$$

$$x = \frac{1}{\sqrt{1-\beta_1}} z_1 - \frac{\beta_1}{\sqrt{1-\alpha_1}\sqrt{1-\beta_1}} g_1[z_1, \phi_1]$$