

## Deep Learning

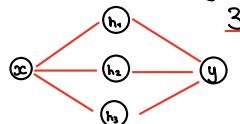
### Shallow Neural Networks (Single hidden layer)

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

#### Single Input to Single Output

$$y = f[\underline{x}, \theta] = \theta_0 + \theta_1 a[\theta_{10} + \theta_{11} x_1] + \theta_2 a[\theta_{20} + \theta_{21} x_1] + \theta_3 a[\theta_{30} + \theta_{31} x_1]$$

Input:  $x_1$ , output:  $y$ , Parameters  $\theta = \{\theta_0, \theta_1, \theta_2, \theta_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$



3 Steps : 1) Compute linear junctions :

$$z_1 = \theta_{10} + \theta_{11} x_1, \quad z_2 = \theta_{20} + \theta_{21} x_1, \quad z_3 = \theta_{30} + \theta_{31} x_1$$

2) Apply activation functions,  $a$

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

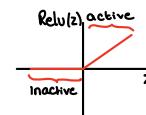
3) Weight three activations and apply offset

$$y = \theta_0 + \theta_1 h_1 + \theta_2 h_2 + \theta_3 h_3$$

Representation with extra node for biases.

$a$  - activation function

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



Training: Given training dataset  $\{\underline{x}_i, y_i\}^T$

Define loss function  $L[\theta]$  to measure

Success/accuracy network.

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

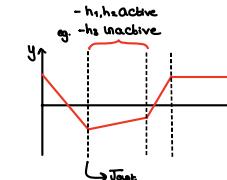
#### Hidden Units / Joints

Using the example of ReLU, although similar for other activation functions.

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

Each hidden units  $\rightarrow$  1 joint

3 hidden units  $\rightarrow$  3 joint  $\rightarrow$  4 linear regions.



#### Universal Approximation Theorem

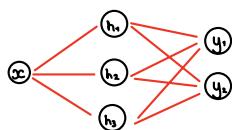
Neural network of 0 hidden units: 0+1 linear regions

$$h_d = a[\theta_{d0} + \theta_{di} x_i] \rightarrow y = \theta_0 + \sum_{d=1}^D \theta_d h_d$$

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 specified continuous function on a subset of  $\mathbb{R}^n$  to arbitrary accuracy.

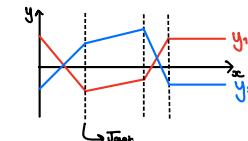
#### Multivariate Outputs



Share the same hidden units/functions:

$$\begin{aligned} y_1 &= \theta_{10} + \theta_{11} h_1 + \theta_{12} h_2 + \theta_{13} h_3 \\ y_2 &= \theta_{20} + \theta_{21} h_1 + \theta_{22} h_2 + \theta_{23} h_3 \end{aligned}$$

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



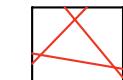
#### Multivariate Inputs:



$$\begin{aligned} h_1 &= a[\theta_{10} + \theta_{11} x_1 + \theta_{12} x_2] \\ h_2 &= a[\theta_{20} + \theta_{21} x_1 + \theta_{22} x_2] \\ h_3 &= a[\theta_{30} + \theta_{31} x_1 + \theta_{32} x_2] \end{aligned} \quad \left. \begin{aligned} y &= \theta_0 + \theta_1 h_1 + \theta_2 h_2 + \theta_3 h_3 \end{aligned} \right\}$$

2D input - 3D input/output Space

Becomes a plane, Clipped to Zero along line



Continuous at boundary - linear planes

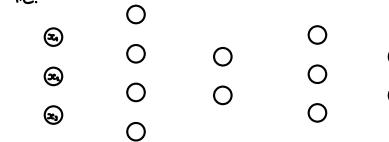
#### Multi Input - Multi outputs (Single hidden layer)

Share joint/ boundaries. but different Scaling.

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

$$h_d = \alpha \left[ \Theta_{d0} + \sum_{i=1}^{D_i} \Theta_{di} x_i \right] \rightarrow y_j = \phi_j \circ \sum_{d=1}^{D_o} \phi_{jd} h_d$$

i.e.



Input  $x$       Hidden 1,  $D_1 = 4$       Hidden 2,  $D_2 = 2$       Hidden 3,  $D_3 = 3$       Output Layer,  $D_o = 2$

## Terminology

Biases -  $\gamma$ -offsets

Fully Connected Network all neurons between layers connected

Capacity  $\approx$  No. hidden units

Weights - gradients

Feed Forward Network no loops

Depth  $\approx$  No. hidden layers

Shallow NN - 1 hidden layer

Pre-activation - Values before activation,  $Z \rightarrow \Theta_{d0} + \sum_{i=1}^{D_i} \Theta_{di} x_i$

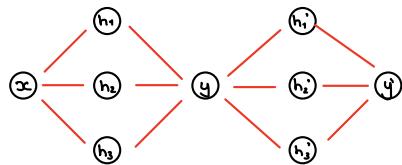
Width  $\approx$  No. hidden units per layer

Deep NN - 1 $L$  hidden layers

Activations - Values post activations,  $a(Z) \rightarrow a \left[ \Theta_{d0} + \sum_{i=1}^{D_i} \Theta_{di} x_i \right]$

## Deep Neural Network (more than 1 hidden layer)

### Example 1: two Connected Shallow Networks



Network 1:  $x \rightarrow y$

$$h_1 = \alpha[\Theta_{10} + \Theta_{11}x]$$

$$h_2 = \alpha[\Theta_{20} + \Theta_{21}x]$$

$$h_3 = \alpha[\Theta_{30} + \Theta_{31}x]$$

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

Network 2:  $y \rightarrow y'$

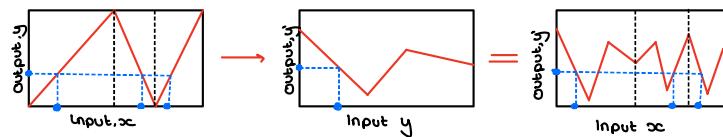
$$h_1' = \alpha[\Theta'_{10} + \Theta'_{11}y]$$

$$h_2' = \alpha[\Theta'_{20} + \Theta'_{21}y]$$

$$h_3' = \alpha[\Theta'_{30} + \Theta'_{31}y]$$

$$y' = \phi'_0 + \phi'_1 h_1' + \phi'_2 h_2' + \phi'_3 h_3'$$

Consider two independent linear maps:



Non 1-to-1 mapping:

In the example above: Network 1  $\rightarrow$  three alternating regions of +ve/-ve slopes

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

↳ 2<sup>nd</sup> Mapping:  $y \rightarrow y'$  applied three times.  $\rightarrow$  2<sup>nd</sup> Network mapping duplicated 3 times.

$\approx$  9 linear regions. (duplicates/stretched/reversed)

### 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.

$$h_1 = \alpha[\Theta'_{10} + \Theta'_{11}h_0 + \Theta'_{12}h_1 + \Theta'_{13}h_2 + \Theta'_{14}h_3]$$

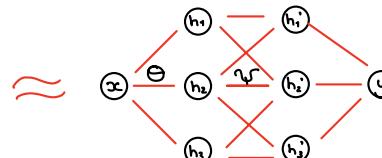
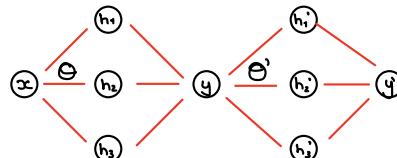
$$\text{or re-written: } h_1 = \alpha[\mathcal{W}_{10} + \mathcal{W}_{11}h_0 + \mathcal{W}_{12}h_1 + \mathcal{W}_{13}h_2 + \mathcal{W}_{14}h_3]$$

$$\text{Similarly: } h_2 = \alpha[\mathcal{W}_{20} + \mathcal{W}_{21}h_0 + \mathcal{W}_{22}h_1 + \mathcal{W}_{23}h_2 + \mathcal{W}_{24}h_3]$$

$$h_3 = \alpha[\mathcal{W}_{30} + \mathcal{W}_{31}h_0 + \mathcal{W}_{32}h_1 + \mathcal{W}_{33}h_2 + \mathcal{W}_{34}h_3]$$

$$\mathcal{W}_{10} = \Theta'_{11}\phi_0, \mathcal{W}_{11} = \Theta'_{11}\phi_1, \dots$$

Thus:



Single Input:  $x$

1<sup>st</sup> Hidden Layer  $h_1 = \alpha[\Theta_{10} + \Theta_{11}x]$

Activations:  $h_2 = \alpha[\Theta_{20} + \Theta_{21}x]$

$h_3 = \alpha[\Theta_{30} + \Theta_{31}x]$

2<sup>nd</sup> Hidden Layer

activations  $h_1 = \alpha[\mathcal{W}_{10} + \mathcal{W}_{11}h_1 + \mathcal{W}_{12}h_2 + \mathcal{W}_{13}h_3]$

$h_2 = \alpha[\mathcal{W}_{20} + \mathcal{W}_{21}h_1 + \mathcal{W}_{22}h_2 + \mathcal{W}_{23}h_3]$

$h_3 = \alpha[\mathcal{W}_{30} + \mathcal{W}_{31}h_1 + \mathcal{W}_{32}h_2 + \mathcal{W}_{33}h_3]$

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

Pre-activation - Simply a shallow network with 3 outputs.

Post activation, adds new joints to each

## Matrix Notation

$$1) \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \alpha \begin{bmatrix} \Theta_{10} \\ \Theta_{20} \\ \Theta_{30} \end{bmatrix} + \begin{bmatrix} \Theta_{11} \\ \Theta_{21} \\ \Theta_{31} \end{bmatrix} x$$

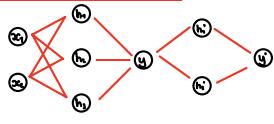
$$\begin{bmatrix} h_1' \\ h_2' \\ h_3' \end{bmatrix} = \alpha \begin{bmatrix} \mathcal{W}_{10} \\ \mathcal{W}_{20} \\ \mathcal{W}_{30} \end{bmatrix} + \begin{bmatrix} \mathcal{W}_{11} & \mathcal{W}_{12} & \mathcal{W}_{13} \\ \mathcal{W}_{21} & \mathcal{W}_{22} & \mathcal{W}_{23} \\ \mathcal{W}_{31} & \mathcal{W}_{32} & \mathcal{W}_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

$$y' = \phi'_0 + [\phi'_1 \ \phi'_2 \ \phi'_3] \begin{bmatrix} h_1' \\ h_2' \\ h_3' \end{bmatrix}$$

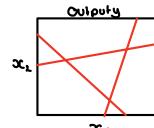
$$2) h = \alpha[\Theta_0 + \Theta x] \quad h' = \alpha[\mathcal{W}_0 + \mathcal{W} h] \quad y' = \phi'_0 + \phi' h'$$

$h, \Theta_0, h', \mathcal{W}_0$   $\rightarrow$  Vectors for each layers activations/biases.  $\mathcal{W}$  - Matrix. In this example (Single input/output):  $\Theta_0$  - Scalar  $\phi'$  - Vector  $\Theta$  - Vector

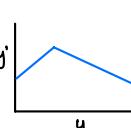
## 2 Dimensional Input



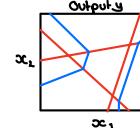
## Network 1



## 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.  $h_k = a[\beta_{k-1} + \Omega_{k-1} h_{k-1}]$

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

## Sizes:

For DNN with  $K$  layers  $y = f[x, \theta]$ :  $h_K = a[\beta_{K-1} + \Omega_{K-1} h_{K-1}]$  (last hidden layer)

$\beta_k$ : Vector size  $D_k$  ( $B_k$ -Size  $D_0$ )

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

$\Omega_k$ : Matrix size  $D_{k-1} \times D_k$  ( $\Omega_0$ -Size  $D_1 \times D_0$ )  
( $\Omega_K$ -Size  $D_K \times D_0$ )

## 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

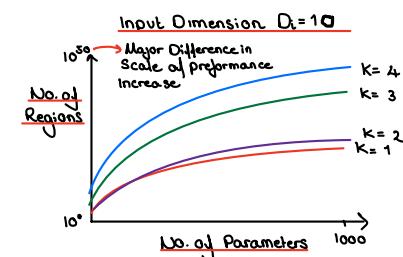
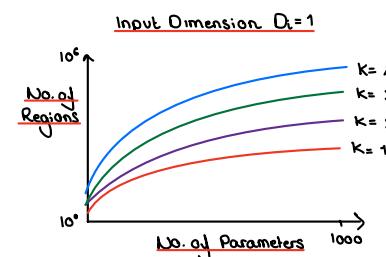
### For a fixed parameter budget:

Deep Neural Networks Create much more complex functions than Shallow Neural Networks

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

### Disadvantages:

DNNs contain complex dependencies and symmetries (folding analogy)  
so although more regions, may only be advantageous for:  
→ similar 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:

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

## Loss/Cost Functions, $L[\theta]$

Returns a single value which describes the disagreement between the model predictions  $\hat{y}[x_i, \theta]$  and true output  $y_i$ :

dependent on:  $L[\theta, \hat{y}[x, \theta], \{x_i, y_i\}_{i=1}^n] \rightarrow L[\theta]$   
Params Model Data

## Types of ML Regression

→ Univariate Regression (one real output)

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

Multivariate Regression ( $> 1$  real output)

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

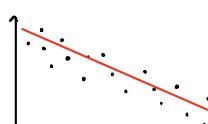
## Probabilistic vs Deterministic Modeling

### Deterministic Machine Learning

→ Simply return prediction  
↳ (Number in regression, relative Probability in classification)  
Note: no uncertainty in own prediction

No quantification of uncertainty

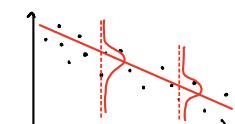
Prone to overconfidence



### Probabilistic Machine Learning

→ Model outputs a probability distribution of outputs  
↳ In fact 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  $x$

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

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

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

Assuming total likelihood

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[ \prod_{i=1}^n P(y_i|x_i, \theta) \right] = \underset{\theta}{\operatorname{argmax}} \left[ \prod_{i=1}^n P(y_i|\theta_i) \right] = \underset{\theta}{\operatorname{argmax}} \left[ \prod_{i=1}^n P(y_i|f[x_i, \theta]) \right]$$

Assume data is i.i.d - Optimise distributions over samples.

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

ii) How the parameters  $\theta$  change with  $x$ ,  $\theta = f[x; \theta]$  (machine learning model)

### Log-Likelihood Criterion

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

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

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

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[ -\sum_{i=1}^n \log [P(y_i|f[x_i, \theta])] \right]$$

### Inference with Probabilistic Machine Learning

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

Must maximise distribution:

$$\hat{y} = \underset{y}{\operatorname{argmax}} [P(y|f[x; \theta])]$$

Often achievable directly from parameters,  $\theta$   
i.e. Normal  $\rightarrow$  Mean,  $\mu$

### Summary of Loss Junctions

1) Choose underlying probability distribution,  $P(y|\theta)$ , of output  $y$  given  $\theta$

2) Set ML model to map  $\text{input} \rightarrow x$   $\theta = f[x; \theta]$

3) Train model by minimising  $L(\theta)$  over training data

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

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

### Example 1: Univariate Regression

↳ Single/Scalar Output ( $y \in \mathbb{R}$ ) from Input

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

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

3) Define Loss function and train by minimising

$$L[\phi] = -\sum_i^{\infty} \log [P(y_i | f[x_i, \phi], \sigma^2)] = -\sum_i^{\infty} \log \left[ \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left[ -\frac{(y_i - f[x_i, \phi])^2}{2\sigma^2} \right] \right]$$

$$\hat{\phi} = \arg \min_{\phi} -\sum_i^{\infty} \log \left[ \frac{1}{\sqrt{2\pi\sigma^2}} \right] - \frac{(y_i - f[x_i, \phi])^2}{2\sigma^2} \rightarrow \arg \min_{\phi} \sum (y_i - f[x_i, \phi])^2$$

≈ Least Squares for fixed variance.

4) Inference:  $\hat{y} = \arg \max [P(y | f[x, \hat{\phi}], \sigma^2)]$

For Univariate Normal:  $\hat{y} = f[\hat{x}, \hat{\phi}]$

Note: For Variance (fixed),  $\sigma^2$  (Homoscedastic)

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

### Heteroscedastic Regression

Model both Mean and Variance ( $\mu, \sigma^2$ ) as function of  $x$ .

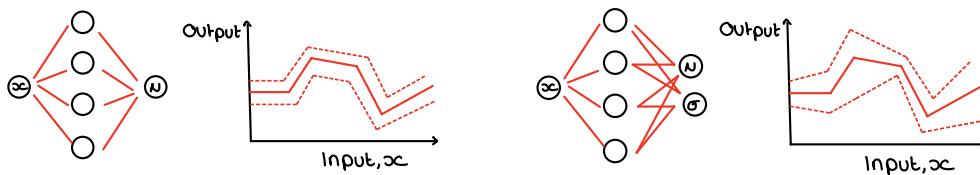
$$\mu = f_1[x, \phi] \quad \sigma^2 = f_2[x, \phi]^2 \rightarrow \text{In fact model std dev and square}$$

to ensure positivity

$$\hat{\phi} = \arg \min_{\phi} \left[ -\sum_i^{\infty} \log \left[ \frac{1}{\sqrt{2\pi f_2[x_i, \phi]^2}} \right] - \frac{(y_i - f_1[x_i, \phi])^2}{2f_2[x_i, \phi]^2} \right] \rightarrow \text{Optimising } \phi \text{ which effect both } f_1 \text{ and } f_2$$

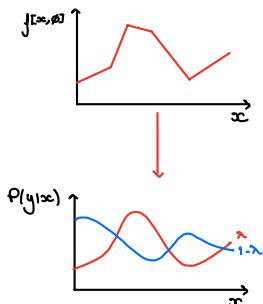
### Univariate (Homoscedastic) vs Multivariate (Heteroscedastic) Regression

↳ 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 for more parameters  
 $\lambda \in [0, 1] \quad \lambda = \text{Sig}[f[x, \phi]]$

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

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

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

### Example 3: Multi Class Classification

1) Underlying (binary) distribution:  $P(y=k) = \lambda_k \quad \lambda_k \in [0, 1] ; \sum_k \lambda_k = 1$

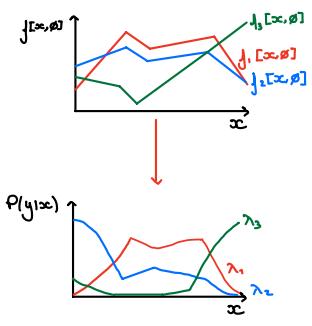
Categorical Distribution:

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

2) ML model:  $P(y=k|x) = \text{Softmax}_K[z] = \frac{\exp[z_k]}{\sum_{k=1}^K \exp[z_k]}$

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

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



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

## Multiple Outputs

Either - Define a multivariate probability distribution and model with NN

or - Treat each prediction as independent

Treat each output  $y_j$  as independent:

(product of univariate terms for  $y_j \in \mathcal{Y}$ )

$$P(y|f(x, \theta)) = \prod_j P(y_j|f_j(x, \theta))$$

↳  $\mathcal{Y}$  = Set of network outputs.

↳ Parameter distribution  $y_j$

Negative Log Likelihood:

$$L[\theta] = -\sum_i \log [P(y_i|f(x_i, \theta))] = -\sum_i \log [P(y_i|f_j(x_i, \theta))]$$

↳ Can use multiple prediction types and treat errors as independent

ie: two different loss func / distributions Summed.

## Cross Entropy Loss

↳ Equivalent alternative to negative log-likelihood.

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

Measure of Distance:

$$D_{KL}[q_\pi||P] = \int_{-\infty}^{\infty} q_\pi(y) \log [q_\pi(y)] dy - \int_{-\infty}^{\infty} q_\pi(y) \log [P(y|\theta)] dy$$

divergence

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

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

↳ Dirac Delta Function

and model distribution

$$P(y) = P(y|\theta)$$

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

No  $\theta$  dependence  
↳ 'Cross entropy'  
- Amount of uncertainty in one distribution given other

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

$$\rightarrow \underset{\theta}{\operatorname{argmin}} \left[ - \sum_i \log [P(y_i|\theta)] \right]$$

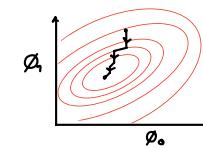
$$\text{In probabilistic machine learning: } \theta = \int [x_i, \theta] \rightarrow \hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[ - \sum_i \log [P(y_i|f_j(x_i, \theta))] \right] \rightarrow \text{Thus shown equivalence to neg log likelihood Criterion.}$$

## Optimisation

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

The loss of a network is dependent on network parameters

Learn by → Iteratively adjusting  $\theta$  based on gradients of loss wrt  $\theta$



## Gradient Descent:

Step 1: Calculate gradients of loss wrt to parameters.

$$\frac{\partial L}{\partial \theta} = \begin{bmatrix} \frac{\partial L}{\partial \theta_0} \\ \frac{\partial L}{\partial \theta_1} \end{bmatrix}$$

↳ defines 'uphill' direction

Step 2: Update parameters

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\partial L}{\partial \theta}$$

↳ moves downhill (-ve sign)

$\alpha$  - Learning rate  
(magnitude of change)

Considering Loss for data set  $\{\mathbf{x}_i, y_i\}$

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

$$\frac{\partial L}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{i=1}^n l_i = \sum_{i=1}^n \frac{\partial l_i}{\partial \theta}$$

Local Minima:

Linear regression

→ Convex loss function - well defined global minimum

Convexity → guaranteed 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 negative gradient (Search direction),  $d_k$

2) Evaluates loss at different Step sizes along Search direction:  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k d_k$ ,  $f(\mathbf{x}_{k+1})$

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)

$$\mathbf{\theta}_{t+1} = \mathbf{\theta}_t - \alpha \sum_{i \in B} \frac{\partial l_i[\theta_t]}{\partial \theta}$$

↳ each subset has its own loss surface

↳ Restricted to batch

→ 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  $\alpha$  decreases → Early: Explore parameter space

for each epoch.      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

$$\mathbf{m}_{t+1} = \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in B} \frac{\partial l_i[\theta_t]}{\partial \theta}$$

→ Accelerates in direction of consistent gradients

Dampens oscillatory behaviour (ie valleys)

Nesterov Accelerated Momentum

Treats momentum term as a prediction of next move.

NAM computes gradients at this predicted point

$$\mathbf{m}_{t+1} = \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in B} \frac{\partial l_i[\theta_t - \alpha \beta \cdot \mathbf{m}_t]}{\partial \theta}$$

average from previous  
Correction factor  
gradient at 'hypo' point

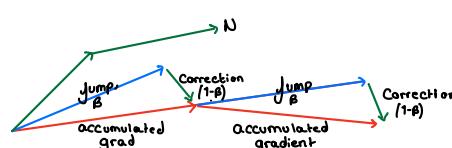
$$\mathbf{\theta}_{t+1} = \mathbf{\theta}_t - \alpha \cdot \mathbf{m}_{t+1}$$

Nesterov Diagram:

↳ Make big jump in direction of previous accumulated gradient ( $\mathbf{x}_k \rightarrow \mathbf{x}_{k+1}$ )

Measure gradient in updated position (Lookahead Gradient) ( $\mathbf{x}_{k+1}$ )

Make Correction to overall gradient and use this ( $\mathbf{x}_k \rightarrow \mathbf{x}_{k+1}$ )



## Adam

Solved issues with fixed learning rate

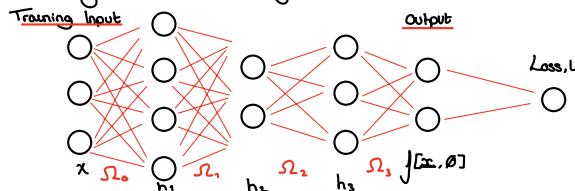
↳ Slow in small gradients, fast in large gradients with no knowledge of curvature.  $\left(\frac{\partial L[\theta_t]}{\partial \theta}\right)$  acts as a proxy for variance using minibatch.

$$m_{t+1} = B \cdot m_t + (1-B) \nabla L[\theta_t] \quad \tilde{m}_{t+1} = \frac{m_{t+1}}{1-B^{t+1}}$$

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

$$\theta_{t+1} = \theta_t - \alpha \frac{\tilde{m}_{t+1}}{\sqrt{\tilde{v}_{t+1}} + \epsilon}$$

## 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  $\Omega_0$  effects loss ( $\frac{\partial L}{\partial \Omega_0}$ )

↳ 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  $J[\mathbf{x}; \theta]$  with inputs  $\mathbf{x}$ ,  $K$  hidden layers

and activation function  $\alpha$ . Individual loss function  $l_i = l[J[\mathbf{x}; \theta], y]$

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  $\beta_k$  and  $\Omega_k$

### Forward Pass: Determine preactivations and activations of each layer

$$j_0 = \beta_0 + \Omega_0 \cdot x_i$$

$$h_k = \alpha[j_{k-1}] \quad k \in \{1, 2, \dots, K\}$$

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

### Backward Pass:

#### Gradient Propagation for Whole layers:

$$\frac{\partial l_i}{\partial j_{k-1}} = \alpha'(j_{k-1}) \odot \Omega_k^T \frac{\partial l_i}{\partial j_k}$$

element-wise multiplication

$\Omega_k^T$  can be thought of as  $\frac{\partial j_k}{\partial h_k} = \frac{\partial}{\partial h_k} (B_k + \Omega_k h_k) = \Omega_k^T$   
 $\alpha'(j_{k-1})$  - Derivative of Activation function

#### Gradient wrt Individual Biases

$$\frac{\partial l_i}{\partial \beta_k} = \frac{\partial l_i}{\partial j_k} \quad k \in \{1, 2, \dots, K\}$$

#### Gradient wrt Individual Weights

$$\frac{\partial l_i}{\partial \Omega_k} = \frac{\partial l_i}{\partial j_k} h_k^T \quad k \in \{1, 2, \dots, K\}$$

#### Exceptions for first layer:

$$\frac{\partial l_i}{\partial \beta_0} = \frac{\partial l_i}{\partial j_0}$$

$$\frac{\partial l_i}{\partial \Omega_0} = \frac{\partial l_i}{\partial j_0} \cdot x_i^T$$

In diagrams:

Forward Pass:

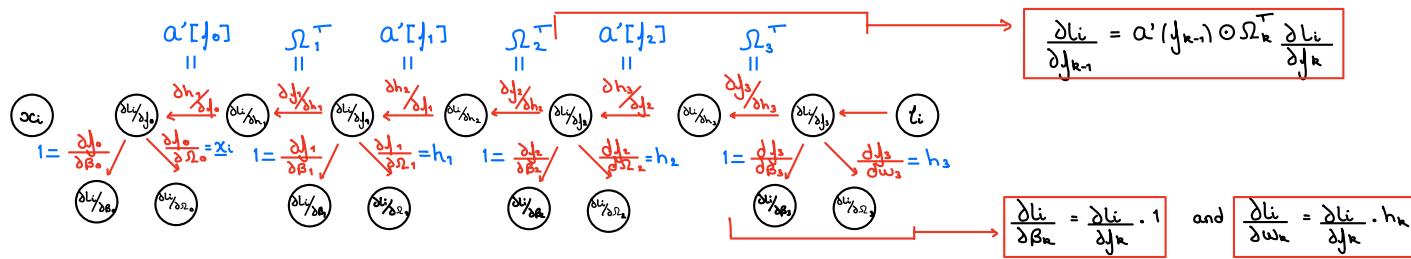


$$j_0 = \beta_0 + \Omega \cdot x_i$$

$$h_k = \alpha[j_{k-1}] \quad k \in \{1, 2, \dots, K\}$$

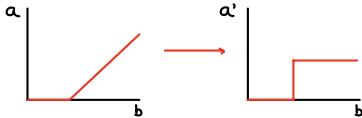
$$j_k = \beta_k + \Omega_k \cdot h_k \quad k \in \{1, 2, \dots, K\}$$

Backwards Pass:



ReLU activation function - back propagation

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



Forward vs Reverse Mode

Forward Mode:

Primals →  
Derivatives

Propagates Derivatives forwards

Each intermediate value differentiated wrt input variables

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

Reverse Mode:

Primals →  
Derivatives.

Backwards pass propagates from outputs to inputs

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

Parameter Initialisation

Forward pass of a general NN:

$$\begin{aligned} j_k &= \beta_k + \Omega_k \cdot h_k \\ &= \beta_k + \Omega_k \cdot \alpha[j_{k-1}] \end{aligned}$$

Initialisation:  $\beta_k = 0 \quad \forall k$   
 $\Omega_k \sim N(0, \sigma^2)$

Qualitatively:

For very Small Variance,  $\sigma^2$ :

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

Additionally, ReLU clips negative values so  $\sim 1/2$ 's width of  $j_{k-1} \rightarrow h_k$

Overall activations shrink across layers → Vanishing activations

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

For very Large Variance,  $\sigma^2$ :

With large weights, weighted sum likely very large.

ReLU clips negative values but if  $\sigma^2$  sufficient large:

Activations grow across layers → Exploding activation

Similarly for backwards pass

$$\frac{\partial L_i}{\partial j_{k-1}} = \alpha'(j_{k-1}) \odot \Omega_k^T \frac{\partial L_i}{\partial j_k}$$

Each gradient update - multiplying by  $\Omega_k^T$

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

Exploding Gradients → Exploding Gradients → Uncontrolled / unstable learning

## Quantitatively:

Considering layer preactivations  $j \rightarrow j'$  with Dimensions  $D_h$  and  $D_n$ :

$$h = a[j]$$

Preactivation in layer  $j$  have variance  $\sigma_{j,i}^2$

$$j' = \beta + \sum h$$

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

### Mean + Variance of $j'$

Mean:  $E[j']$

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

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

$$N_{j,i} = 0$$

$$\sigma_{j,i}^2 = \frac{1}{2} D_h \sigma_n^2 \sigma_j^2$$

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

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

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

$$= \sigma_n^2 \sum_{j=1}^{D_h} E[h_j^2] = \sigma_n^2 \sum_{j=1}^{D_h} \frac{\sigma_j^2}{2} = \frac{1}{2} D_h \sigma_n^2 \sigma_j^2$$

Assume  $j_i$  are Symmetrically distributed about Zero  
 $\therefore$  ReLU will Clip  $\frac{1}{2}$  of preactivation -  $\frac{1}{2}$  variance

## He Initialisation

↳ To ensure Variance to be constant. i.e.  $\sigma_j^2 = \sigma_{j,i}^2$

$$\sigma_n^2 = \frac{2}{D_h}$$

Similar by back propagation of gradients:

$$\sigma_n^2 = \frac{2}{D_n}$$

### For non-Square $\Omega$

↳  $D_h \neq D_n$  - no. hidden units in layer differs

$$\text{Must Compromise ie } \sigma_n^2 = \frac{2}{\left(\frac{D_h + D_n}{2}\right)} = \frac{4}{D_h + D_n}$$

## Sources of Error

### Noise:

Inherent uncertainty in true mapping from input to output.

↳ Result of genuine Stochastic Nature, limited precision of measurement, Unobserved latent variables...

Introduces error for a perfectly fitted model - 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  $\sigma^2$ :

expectation of observed values.

$$\text{Expected value } N[x] = E_y[y|x] = \int y|x P(y|x) dy$$

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

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

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

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

$E_y[L(\mathbf{x})]$   $= (f[\mathbf{x}, \emptyset] - N(\mathbf{x}))^2 + 2(f[\mathbf{x}, \emptyset] - N(\mathbf{x})) \underbrace{E[(N(\mathbf{x}) - y[\mathbf{x}])]}_{N(\mathbf{x}) - y[\mathbf{x}] = 0} + \underbrace{E[(N(\mathbf{x}) - y[\mathbf{x}])^2]}_{\text{noise: } \sigma^2}$

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

$\emptyset$  is a function of training dataset  $\mathcal{D} \rightarrow \emptyset[\mathcal{D}]$

Expectation over datasets (thus  $\emptyset$ )

$$f_N[\mathbf{x}] = E_{\mathcal{D}}[f[\mathbf{x}, \emptyset[\mathcal{D}]]]$$

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

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

$E_{\mathcal{D}}[(f[\mathbf{x}, \emptyset] - N(\mathbf{x}))^2]$   $= E_{\mathcal{D}}[(f[\mathbf{x}, \emptyset] - f_N[\mathbf{x}])^2] + 2 \underbrace{E[f[\mathbf{x}, \emptyset] - f_N[\mathbf{x}]]}_{\text{Variance of model}} (f_N[\mathbf{x}] - N(\mathbf{x})) + E_{\mathcal{D}}[(f_N[\mathbf{x}] - N(\mathbf{x}))^2]$

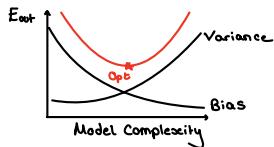
$$f_N[\mathbf{x}] - f_N[\mathbf{x}] = 0$$
$$(f_N[\mathbf{x}] - N(\mathbf{x}))^2$$

Overall:

$$E_{\mathcal{D}}[E_y[L(\mathbf{x})]] = \underbrace{E_{\mathcal{D}}[(f[\mathbf{x}, \emptyset[\mathcal{D}]] - f_N[\mathbf{x}])^2]}_{\substack{\text{Variance} \\ (\text{Finite Data} - \text{Infinite Data}) \\ \text{Model}}} + \underbrace{(f_N[\mathbf{x}] - N(\mathbf{x}))^2}_{\substack{\text{Bias} \\ (\text{Infinite Data} - \text{True Model})}} + \underbrace{\sigma^2}_{\text{Noise}}$$

Bias - Variance Tradeoff:

For finite data regime:



By Increasing Complexity

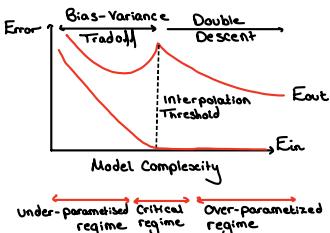
Greater ability to: Represent underlying function  $\rightarrow$  Decrease bias

Overfit to Statistical Fluctuations  $\rightarrow$  Increase Variance

Smoother So

Smoothen So

Double Descent



Interpolation threshold:

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

Surprise: Test error decreases as model grows into highly overparametrised (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 to choose among multiple fits - not able to find smooth, generalisable function

## Overparametrised Models

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

↳ 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 performances)

→ Reduce overfitting - learning statistical peculiarities

Improve generalisation - for unconstrained areas of no training data.

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

Normal Loss Term

Penalty Loss

$\lambda$  - hyperparameter of regularisation strength

$g(\theta)$  - large scalar value for less preferable parameters.

## Probabilistic Interpretation

The regularisation term is equivalent to a Prior,  $Pr(\theta)$

↳ encompasses 'knowledge / information' on parameters

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[ \sum_{i=1}^n l_i(x_i, y_i) \right]$$

$$\underset{\theta}{\operatorname{argmax}} \left[ \sum_{i=1}^n l_i(x_i, y_i) \Pr(\theta) \right]$$

Maximum a posteriori - MAP

$$\text{With } \log[Pr(\theta)] = \lambda \cdot g(\theta)$$

## L2 - Regularisation (Ridge Regression)

Regularisation term,  $g(\theta) = \sum \theta_i^2$ , L2 Norm

(Typically applied to weights not biases)

Encourages smaller weights

↳ Smoother output function

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left[ \sum_{i=1}^n l_i(x_i, y_i) + \lambda \sum \theta_i^2 \right]$$

## Explanation:

↳ Adding regularisation - must trade off 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 infinitesimal step size (perfect descent):

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

For discrete step size,  $\alpha$ :

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial L}{\partial \theta}$$

Deviates from continuous path.

## Equivalent Loss Junction:

[Seek to find connection  $g_i(\theta)$  to continuous version s.t. gives discrete version]

(use backwards error analysis)

For  $\alpha \rightarrow 0$  step size:

$$\frac{d\theta}{dt} = g(\theta) \rightarrow \frac{d\theta}{dt} \approx g(\theta) + \alpha g_i(\theta)$$

Taylor Expansion,  $\alpha = (\theta - \theta_0)$

$$\theta \approx \theta_0 + \alpha \frac{d\theta}{dt} + \frac{\alpha^2}{2} \frac{d^2\theta}{dt^2} \Big|_{\theta=\theta_0} \rightarrow \theta + \alpha (g(\theta) + \alpha g_i(\theta)) + \alpha^2/2 \left( \frac{\partial g(\theta)}{\partial \theta} \frac{d\theta}{dt} + \alpha \frac{\partial g_i(\theta)}{\partial \theta} g(\theta) \right) \Big|_{\theta=\theta_0}$$

→

$$\theta + \alpha (g(\theta) + \alpha g_i(\theta)) + \frac{\alpha^2}{2} \left( \frac{\partial g(\theta)}{\partial \theta} g(\theta) + \alpha \frac{\partial g_i(\theta)}{\partial \theta} g(\theta) \right) \Big|_{\theta=\theta_0}$$

## Equivalent Loss Junction:

$$L_{\text{discrete}}[\theta] = L[\theta] \rightarrow \frac{\alpha}{4} \left\| \frac{\partial L}{\partial \theta} \right\|^2$$

arrives at same place

as discrete

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

$$\approx \underset{\beta=0}{\left. \left( \emptyset + \alpha g[\emptyset] + \alpha^2 \left( g_i[\emptyset] + \frac{1}{2} \frac{\partial g[\emptyset]}{\partial \beta} g[\emptyset] \right) \right) \right|} \quad (\text{remove } \mathcal{O}(\alpha^2))$$

$$\emptyset_0 + \alpha g[\emptyset_0] \quad \text{For equally must } = 0$$

Discrete case

$$g_i[\emptyset] = -\frac{1}{2} \frac{\partial g[\emptyset]}{\partial \beta} g[\emptyset] \quad \rightarrow \quad \frac{\partial \emptyset}{\partial t} = -\frac{\partial L}{\partial \emptyset} - \frac{\alpha}{2} \left( \frac{\partial^2 L}{\partial \emptyset^2} \right) \frac{\partial L}{\partial \emptyset}$$

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

Implicit Regularisation: Stochastic GD

$$L_{SGD}[\emptyset] = L[\emptyset] + \frac{\alpha}{4} \left\| \frac{\partial L}{\partial \emptyset} \right\|^2 + \frac{\alpha}{4B} \sum_{b=1}^B \left\| \frac{\partial L_b}{\partial \emptyset} - \frac{\partial L}{\partial \emptyset} \right\|^2$$

↳ B - no. batches

$$L = \frac{1}{B} \sum_i L_i[\emptyset_i, y_i]$$

$L_b$  = Loss of  $b^{\text{th}}$  batch

$$L_b = \frac{1}{|\beta|} \sum_{i \in b} L_i[\emptyset_i, y_i]$$

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 perform 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,  $t(x)$ :

$$f(t(x)) = f(x) \quad \text{ie output unchanged by transformation}$$

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,  $t(x)$  w/:

$$f(t(x)) = t(f(x))$$

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:  $x = [x_1, x_2, \dots, x_n]$

For Kernel Size = 3:

Kernel:  $w = [w_1, w_2, w_3]^T$

$$z_i = w_1 x_{i-1} + w_2 x_i + w_3 x_{i+1} \quad \rightarrow \quad i \rightarrow i+1, z_i \rightarrow z_{i+1} \quad (\text{equivariant wrt translation})$$

### 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-Convolution: 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.

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

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

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

### Dilation rate, D

↳ Introduces gaps between / intersperse Kernel elements

↳ Increases receptive field without no. parameters

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

### 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  $\xrightarrow{\text{Stacked}}$  Feature map or Channel

#### Multi-Channel Systems:

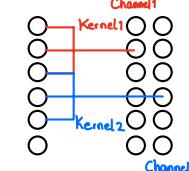
Multiple input Channels → Multiple Output Channels.

Kernels act across multi inputs  $\xrightarrow{\text{Pool}}$  Single Output Channel.

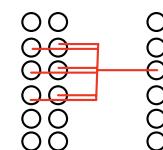
↳ 1 Kernel: N input Channels → 1 output channels

N' kernels: N input Channels → N' output channels

#### Multi-output Channel (Multi Kernels)



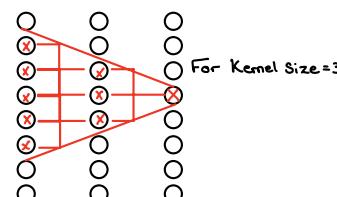
#### Multi Input Channel



↳ 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 KxK region (KxK weights)

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

$h_{i,j}$  - Kernel is translated both horizontally + vertically

2D output Space

## Channels in 2D Networks

↳ Multiple 2D outputs (become 3D input x 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 =  $S$ )

DownSampling happens simultaneously to feature extraction

Maintains Sparsity - Does not prioritize important values.

### 3) Max Pooling

Separates input space into  $S \times S$  blocks → Selects maximum value

Preserves Strongest activations

Helps promote translational invariance at small scales.

### 2) Mean/Average Pooling

↳ Separates input space into  $S \times S$  blocks → Calculates average value

Retains average representation of local region.

## UpSampling

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

### 1) Nearest-Neighbour Upsampling

Each pixel value is duplicated  $S \times S$  times

Results in blocky artifacts / harsh transitions

### 3) Bilinear Interpolation

Estimates missing values by linearly interpolating between nearest values.

### 2) Max unpooling

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

### 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  $1 \times 1$  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 receives previous layers output

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

Think of network as series of nested functions:

$$y = f_3[f_2[f_1[x, \theta_1], \theta_2], \theta_3]$$

## Limitations of Deep Networks

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

Deep networks - performance worsens with more layers

↳ Performance decreases for both test + training sets.

↳ Note: 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.

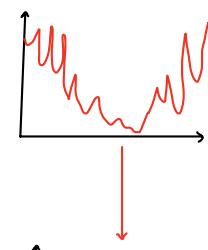
Auto-correlation: In deeper networks gradients become uncorrelated and chaotic

Optimization cannot learn stable/meaningful updates.

Changes in early network layers modify the output in increasingly complex ways

as the network becomes deeper

$$\frac{\partial y}{\partial \theta_1} = \frac{\partial f_2}{\partial \theta_1} \cdot \frac{\partial f_3}{\partial \theta_1} \cdot \frac{\partial f_4}{\partial \theta_1} \rightarrow \text{if parameters } \theta_1 \text{ change - all derivatives in sequence evaluated at different positions (as dependent on } \theta_1 \text{) - updated gradient varies massively}$$

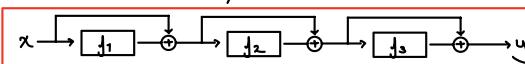


Different failure mode to vanishing/exploding gradients:

↳ Any finite step size - loss surface may be completely different.



## Residual Connections/blocks



$$y = h_2 + f_3[h_2, \theta_3] = x + f_1[x] + f_2[x + f_1[x]] + f_3[x + f_2[x + f_1[x]]]$$

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

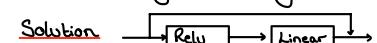
Short Paths - Stable gradients - Learn faster (less shattered gradients)

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

### Order of Operations (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  $\rightarrow$  fl direct path for each layer

If we use He Initialisation: exponential growth in activation variance

↳ Addition of branches - Increases Variance  $\Delta C_{\text{new}} = \Delta C_{\text{old}} + f(\Delta C_{\text{old}})$   $\Delta C_{\text{old}}:$  Variance  $\vee$   $\Delta C_{\text{new}}:$  Variance,  $2V$

Solved with:

## Batch Normalisation

Shifts/rescales each activation  $h_i$  for given hidden unit, St. its mean + variance

are learnable parameters: (standardise hidden features).

For given batch,  $B$  and hidden node:

$$\text{mean } m_n = \frac{1}{|B|} \sum_{i \in B} h_i$$

$$\text{variance: } S_n = \sqrt{\frac{1}{|B|} \sum_{i \in B} (h_i - m_n)^2}$$

Standardise the batch's activations:

$$h_i \leftarrow \frac{h_i - m_n}{S_n + \epsilon} \quad \forall i \in B$$

Adjust with learnable parameters:

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

## During testing:

do not use test batch mean/variance but running average (global distribution) from training

$$\left. \begin{array}{l} N_{\text{run}} \rightarrow (1-\alpha) N_{\text{run}} + \alpha N_B \\ \sigma^2_{\text{run}} \rightarrow (1-\alpha) \sigma^2_{\text{run}} + \alpha \sigma^2_B \end{array} \right\} \quad \hat{\sigma}_i = \frac{\sigma_i - N_{\text{run}}}{\sqrt{\sigma^2_{\text{run}} + \epsilon}}$$

## 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 propagation

↳ In Res NN: variance still increases (additional source variance)

Becomes linear increase.

(initially later layers have smaller effect - tunable  $\gamma$ )

## Higher Learning Rates:

↳ Smoother loss surface and gradients

## Regularisation

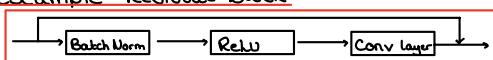
↳ injects batch dependent noise ( $m_n, S_n$ )

Improves generalisation.

## Initialisation

Initialisation of networks ie. Cauchy assume mean, 0, variance, 1 to avoid blow out.

## Example 'Residual Block'



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

## U-Net/Hourglass Networks

↳ Encoder/Decoder

Residual connections transfer information from encoder  $\rightarrow$  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  $\xrightarrow{\text{Subword tokeniser}}$  into a token.

- 1)  $N$  input tokens  $\longrightarrow T \in \mathbb{R}^{1 \times 1 \times N}$ 
  - $n^{\text{th}}$  Column  $\approx n^{\text{th}}$  token
  - Contains  $1 \times 1 \times 1$ , one hot vector

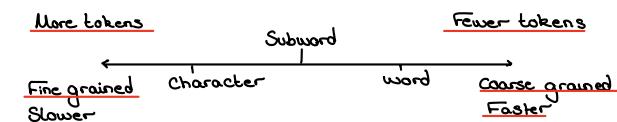
## Embeddings

- 2) Token matrix mapped to Unique (Vector) Word embeddings in lower-dimensional Space (embedding Space) - but not 1 hot vector

- ↳ Vocabulary/Embedding Matrix:  $\Omega_e$

$$x = \Omega_e T$$

## Trade off in tokenisation:



## Byte Pair Encoding (BPE) Tokenisation

- 1) Initialise vocabulary - i.e 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 occurrences with updated token.
- 4) Iterate until desired Vocab size is reached

## Self-Attention:

For  $N$  inputs  $x_1, x_2, \dots, x_N$  (all Dimension:  $D_x$ ) - each representing a word/token

- 1) Recomputes word/tokens representation

$$v_m = \beta_v + \Omega_v x_m \quad \beta_v - \text{Biases}, \Omega_v - \text{Weights}$$

- 2) For each word  $n$

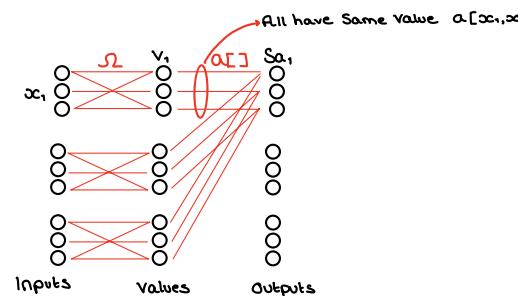
$$s_{a,n} [x_1, \dots, x_N] = \sum_{m=1}^N \alpha [x_m, x_n] v_m \quad \text{Weighted Sum of } v_m, \text{ Weighted by } m^{\text{th}} \text{ relavence to } n \text{ (n's attention term)}$$

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

Attention Weights  $\xrightarrow{\text{Summed input weights.}}$   $\xrightarrow{\text{output } n}$

$n^{\text{th}}$  weights  $\alpha [x_m, x_n]$  - non negative and sum to one

- ↳ Each word's relavence to all outputs sum to unity



## Determining attention weights:

Queries:  $q_{n,m} = \beta_q + \Omega_q x_m$  - word asking for information

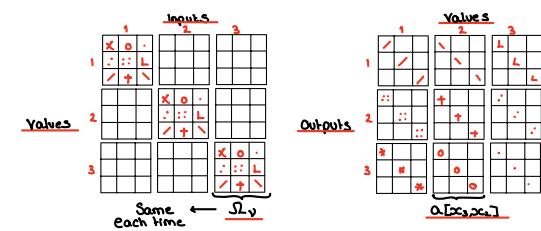
Keys:  $k_{m,n} = \beta_k + \Omega_k x_n$  - word providing information

$\Omega_q, \Omega_k$  - weight matrices

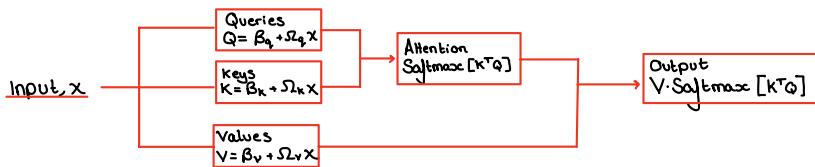
$$\alpha [x_m, x_n] = \text{Softmax}_m [k_m^T q_n] = \frac{\exp [k_m^T q_n]}{\sum_{m=1}^N \exp [k_m^T q_n]}$$

↳ Focussed 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)  $Sa[x] = V[x] \cdot \text{Softmax} \left[ \frac{K[x]^T Q[x]}{\sqrt{D_q}} \right]$

$\downarrow$   
Maintains unit variance

$D_q \cdot \text{Dimensionality of vectors}$

## Intuition behind Q, K and V

Queries (Q) - The Search Criteria for each word in the context of a sentence

↳ (which words are relevant to current word).

Keys (K) - Identifiers that allow words to be matched to queries.

↳ (represent attention a word should receive from query)

Dot Product (QK<sup>T</sup>) - 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:

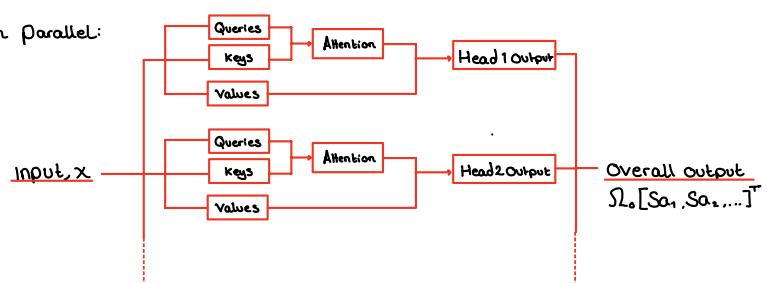
Multihed Self-attention: multiple mechanisms can be applied in parallel:

$h^{th}$  SA mechanism / head:

$$V_h = \beta_{v,h} + \omega_{v,h} x$$

$$Q_h = \beta_{q,h} + \omega_{q,h} x \quad \rightarrow \quad Sa_{h,n}[x] = V_h \cdot \text{Softmax} \left[ \frac{K_h^T Q_h}{\sqrt{D_q}} \right]$$

$$K_h = \beta_{k,h} + \omega_{k,h} x$$



## Multi Heads (Global vs Local)

For input dimension:  $N \times D$   $N$  - no. tokens  $D$  - dimension of embedding space.

With  $H$  head: Each may focus on Subset of embedding space ie  $\text{Dim}(\text{Keys}) \approx D/H \times N$

↳ All Samples but different feature space.

"Similar to CNNs": 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  $TC$  - encodes positional information about input sequence with unique columns.

i.e. 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  $TC_{a,b}$

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

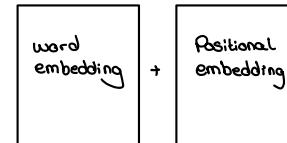
## Sinusoidal Encoding

$\text{Sin}\left(2\pi \frac{n}{d}\right)$   $n$  - position index of token

$d$  - wavelength for each embedding dim

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

Add Sinusoidal 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 (Sentiment 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 Sentence 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 categories i.e. (person, organisation, location)

Passed through Softmax function (multi-class classification).

## Decoder (GPT-3)

↳ Role to generate the next token in a Sequence

Masked Self Attention:

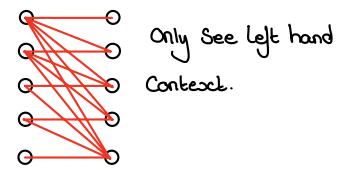
To only allow Context (beforehand) to the masked item.

Attention mechanism by Setting future Context Scores to  $-\infty$  before applying Softmax

$Q_{ji} \cdot K_{j>i} = -\infty$  (keys to RHS of query.)

↳ Softmax:  $\exp(q, k) = 0$ .

Masked Attention



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 Auto encoders

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

VAE - Neural architecture designed to help learn

Final Model  $P(x)$  - Non linear Latent variable model

Latent Variable Models:

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

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

Recover  $P(x)$  from Marginalisation of  $z$ :

$$P(x) = \int P(x, z) dz \rightarrow P(x) = \int P(x|z) P(z) dz$$

Typically Conditional representation

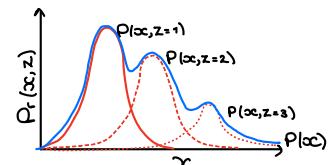
$$P(x, z) = P(x|z) P(z)$$

Prior:  $P(z)$

Likelihood:  $P(x|z)$

Relatively Simple  $P(x|z)$ ,  $P(z)$

Can form complex  $P(x)$ .



Nonlinear latent variable models:

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

Prior,  $P(z)$ : Standard multi-variate normal.

$$P(z) = \text{Norm}_z [0, I]$$

Likelihood,  $P(x|z, \theta)$ : Normally distributed.

$$P(x|z, \theta) = \text{Norm}_x [j[z, \theta], \sigma^2 I]$$

Mean: Non linear function of latent variable

$j[z, \theta]$  → Described by Deep Network, parameters  $\theta$ .

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

$$\Pr(x|\emptyset) = \int \Pr(x, z|\emptyset) dz = \int \Pr(x|z, \emptyset) \Pr(z) dz \\ = \int \text{Norm}_{\infty} [j[z, \emptyset], \sigma^2 I] \cdot \text{Norm}_z [0, I] dz.$$

Summary: Infinite weighted sum of spherical gaussians.

Weights:  $\Pr(z)$

Means:  $j[z, \emptyset]$

Generation from latent variable model (ancestral Sampling):

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

Training:

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

Assuming variance term is known:

$$\hat{\emptyset} = \arg \max_{\emptyset} \left[ \sum_{i=1}^T \log [\Pr(x_i|\emptyset)] \right]$$

$$\Pr(x_i|\emptyset) = \int \text{Norm}_{\infty} [j[z, \emptyset], \sigma^2 I] \cdot \text{Norm}_z [0, I] dz$$

Intractable: No closed form/ easy evaluation.

Evidence Lower Bound (ELBO) on log-likelihood:

Used as proxy for likelihood - depends on additional parameters  $\Theta$ .

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:  $g = \log$

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

$$\log \left[ \int \Pr(y) y dy \right] \geq \Pr(y) \log[y] dy$$

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

Using Jensen's Inequality for ELBO

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

↓ introduce arbitrary probability distribution  $q_z(z)$ .

$$= \log \left[ \int q_z(z) \frac{\Pr(x, z|\emptyset)}{q_z(z)} dz \right] \geq \int q_z(z) \log \left[ \frac{\Pr(x, z|\emptyset)}{q_z(z)} \right] dz \quad \text{Jensen's Inequality}$$

$$\text{ELBO}[\emptyset, \Theta] = \int q_z(z|\emptyset) \log \left[ \frac{\Pr(x, z|\emptyset)}{q_z(z|\emptyset)} \right] dz$$

Introduces  $\Theta$  - parameters of  $q_z(z)$  distribution.

Learning: Must maximise ELBO as both a function of  $\emptyset$  and  $\Theta$ .

→ Adjusting  $\Theta$  - Lower bound distance from log likelihood. ↗

Adjusting  $\emptyset$  - Move along ELBO ↙

## Tightness of Elbo:

For a fixed,  $\emptyset$ : Elbo is tight when  $\text{ELBO} = \text{log likelihood}$ . i.e. optimal  $q_z(z|\theta)$ .

$$\begin{aligned} \text{ELBO}[\theta, \emptyset] &= \int q_z(z|\theta) \left[ \frac{\Pr(x|z, \emptyset)}{q_z(z|\theta)} \right] dz = \int q_z(z|\theta) \log \left[ \frac{\Pr(z|x, \emptyset) \Pr(x|\emptyset)}{q_z(z|\theta)} \right] dz \\ &= \underbrace{\int q_z(z|\theta) \log [\Pr(x|\emptyset)] dz}_{\text{no } z \text{ dependence}} + \underbrace{\int q_z(z|\theta) \log \left[ \frac{\Pr(z|x, \emptyset)}{q_z(z|\theta)} \right] dz}_{\text{KL Divergence}} \\ &= \log \Pr(x|\emptyset) dz - D_{\text{KL}}[q_z(z|\theta) \parallel \Pr(z|x, \emptyset)] \end{aligned}$$

KL Divergence - measures 'distance' between distributions

ie:  $q_z(z|\theta) = \Pr(z|x, \emptyset) \rightarrow \text{KL Divergence} = 0$   
 ↳ posterior distribution over  $L \vee z$

## Alternative expression:

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

$$\text{ELBO}[\theta, \emptyset] = \int q_z(z|\theta) \log [\Pr(x|z, \emptyset)] dz - D_{\text{KL}}[q_z(z|\theta) \parallel \Pr(z)]$$

## Variational Approximation:

Assumption:  $q_z(z|\theta) \approx \text{Multivariate Normal Dist - mean, } N, \text{ Covariance, } \Sigma$

Ideal choice for  $q_z(z|\theta)$  was posterior  $\Pr(z|x, \emptyset)$  which is data dependent. (by KL Div)

To do same:

$$q_z(z|x, \emptyset) = \text{Norm}_z[g_N[x, \emptyset], g_z[x, \emptyset]]$$

-  $g[x, \emptyset]$  is neural network to predict  $N, z$ .

## Variational Autoencoder:

Build a network that computes ELBO:

$$\text{ELBO}[\theta, \emptyset] = \int q_z(z|x, \emptyset) \log [\Pr(x|z, \emptyset)] dz - D_{\text{KL}}[q_z(z|x, \emptyset) \parallel \Pr(z)]$$

Where  $q_z(z|x, \emptyset) = \text{Norm}_z[g_N[x, \emptyset], g_z[x, \emptyset]]$

$$E[\log[\Pr(x|z, \emptyset)]] \approx \frac{1}{N} \sum \log[\Pr(x|z_i)] \approx \log[\Pr(x|z^*)]$$

↳ Estimated from Single Sample

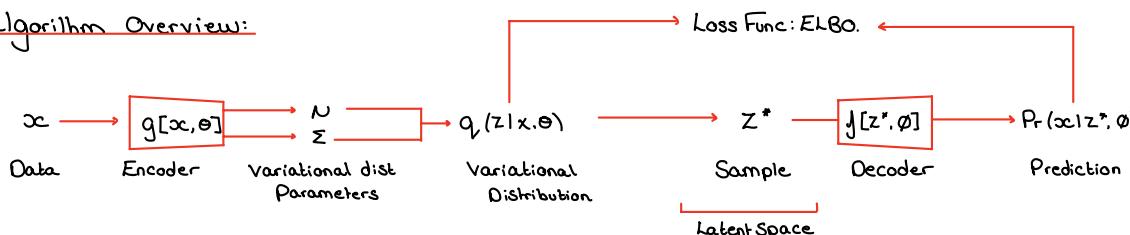
$$\text{ELBO}[\theta, \emptyset] = \log[\Pr(x|z^*, \emptyset)] - D_{\text{KL}}[q_z(z|x, \emptyset) \parallel \Pr(z)]$$

$$q_z(z|x, \emptyset) = \text{Norm}_z[N, \Sigma]$$

$$\Pr(z) = \text{Norm}_z[0, I]$$

$$D_{\text{KL}} = \frac{1}{2} \text{Tr}[\Sigma] + N^T N - D_z - \log[\det[\Sigma]] \quad D_z - \text{Dimensionality of latent space.}$$

## 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  $q_z(z|x, \emptyset)$ )

Decoder,  $f[z, \theta]$

↳ Mapping from latent variable to data.