DEEP LEARNING - OVERVIEW

unsupervised learning / unlabeld data Feature Extraction

supervised learning / labeled data Classification

Model-Selection / Cross-Validation

Data Preparation / Preprocessing → **Define model architecture** → **Train model (with train data)** → **Evaluate model (with validation data)** → **Test model (with test data)**

we need lots of data!

structured data -> tables/databases
unstructured data -> audio/image/text

Shuffle data -> to get same characteristics

Flatten (from 28x28 to 784) -> dimension
more dimension -> more data
(curse of dimensionality)

Split data into 3 subsets -> avoid overfitting
- training (large 98-60% small) -> train model
- validation (1-20%) -> tune hyper-params
- test set (1-20%) -> evaluate model

Normalise all data -> faster convergence
- z-normailsation
  - shift -> improve robustness of learning
  - scale -> faster convergence and accuracy

$$x'^{(i)}_k = \frac{x^{(i)}_k - \mu_k}{\sigma_k} \quad \left| \quad \mu_k = \frac{i}{N}\sum_{k=1}^{N} x^{(i)}_k \right. \quad \left. \sigma^2_k = \frac{i}{N}\sum_{k=1}^{N}\left(x^{(i)}_k - \mu_k\right)^2 \right|$$

- min/max rescaling → [0,1]

$$x'^{(i)}_k = \frac{x^{(i)}_k - min_j\left(x^{(j)}_k\right)}{max_j\left(x^{(j)}_k\right) - min_j\left(x^{(j)}_k\right)}$$

- min/max normalisation → [−1,1]

$$x'^{(i)}_k = 2 * \frac{x^{(i)}_k - min_j\left(x^{(j)}_k\right)}{max_j\left(x^{(j)}_k\right) - min_j\left(x^{(j)}_k\right)} - 1$$

- modify output to 1-hot target (4 -> 00010)

Data Augmentation -> avoid overfitting

$$X = \begin{bmatrix} | & \cdots & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & \cdots & | \end{bmatrix} \in \mathbb{R}^{n_x \times m}$$

$$Y = \begin{bmatrix} y^{(1)} & \cdots & y^{(m)} \end{bmatrix} \in \mathbb{R}^{1 \times m}$$

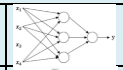
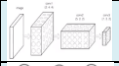
Type
- Sequential (1 to 1 layer, stack)
- Functional (multiple paths/in/out)

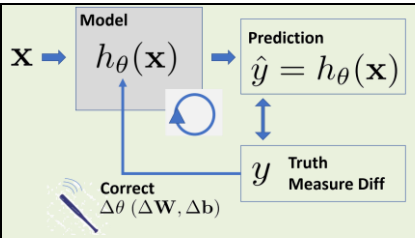
Layers $(\boldsymbol{w} * \boldsymbol{x} + b)$
- Dense, fully connected (# neurons)
- Conv2D (#filter, #channel, filter_size, ...)
- Activation
- MaxPooling2D (pool size)
- Dropout (dropout rate p)
- Flatten
- Batch-Normalisation
- Fall Back layers -> for deep networks
- Bottleneck layer -> reduce params
- GRU (Gated Recurrent Units)
- LSTM (Long Short Term Memory)

Activation functions / Prediction:
-> robustness and performance
- Heavyside
- Sigmoid
- Tanh
- Recti-Linear Unit (ReLU)
- Leaky ReLU (α)
- Exponential Linear Unit (ELU) (α)
- Identity
- Softmax → final layer

Classification / Output
- Binary classification $\begin{cases} 1 = yes \\ 0 = no \end{cases}$
- Multi classification $(K\ classes)$

Sample architetures

Standard NN -> simple application	
CNN -> image recognizing	
RNN -> sequenced data	
Autoencoders -> reproduce input	
Transfer learning -> reuse feature extr	



Computation
Prefer TPU to GPU to CPU

0. Parameter initialisation $\theta = (\boldsymbol{w}, b)$
rand initialisation -> break symmetries
normalize per layer (Xavier)

1. Propagate and make Prediction

$$\hat{y}(\boldsymbol{x}) = round\left(h_\theta(\boldsymbol{x})\right) \in \{0,1\}$$

$\hat{y}(\boldsymbol{x}) = \sigma(w^T x + b)$ -> logistic regression

$$\hat{y}(\boldsymbol{x}) = \underset{l}{argmax} \quad \left\{h_{\theta_l}(x)\right\} \in \{0 \dots K-1\}$$

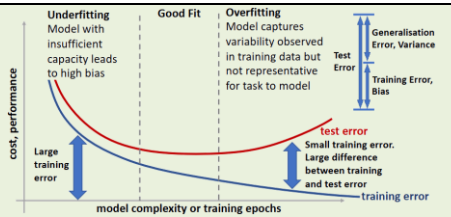
2. Measure Diff
Cost/Loss(Error) Functions
- # of correctly classified samples
- Mean Square Error

$$J_{MSE}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

- Cross-Entropy -> convex function

$$J_{CE}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\log\left(h_{\theta,y^{(i)}}(x^{(i)})\right)$$

-> for classification



High Bias: Model is too simple for data.
High Variance: Model adapts bad to new data.

3. Backprop & update weights
Learning algorithm (Correct)
- Perceptron learning rule

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha * \left(\hat{y}^{(i)} - y^{(i)}\right) * \boldsymbol{x}^{(i)}$$
$$b \leftarrow b - \alpha * \left(\hat{y}^{(i)} - y^{(i)}\right)$$

- **Gradient descent** $\theta_{t+1} \leftarrow \theta_t - \alpha * \nabla_\theta J(\theta_t)$
  - Batch Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
- learning rate α
  too large -> not converge
  too small -> slow convergence
- batch size
- # epochs / iterations
  too small -> not optimal value
  too big -> overfitting

Regularisation -> prefer simple models
- Weight Decay -> add penalty

$$J = J_0 + \lambda\Omega(\mathbf{W})$$

- L1 $\Omega(\boldsymbol{W}) = \|W\|_1 = \sum_{l,k,j}\left|W^{[l]}_{kj}\right|$
- L2 $\Omega(\boldsymbol{W}) = \|W\|_2^2 = \sum_{l,k,j}\left|W^{[l]}_{kj}\right|^2$
- Dropout (temp ignored) -> more robust
- Early Stopping -> avoid overfitting
- Data Augmentation -> more tolerant

Advanced Optimizer
- Momentum / Nesterov (surpass flat, moment β_1)

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + \alpha\nabla J(\theta + \beta_1 m)$$

- RMSprop (adjust α per direction, RMS decay β_2)

$$s \leftarrow \beta_2 s + (1-\beta_2)\nabla J(\theta) \odot \nabla J(\theta)$$
$$\theta \leftarrow \theta - \frac{\alpha}{\sqrt{s+\epsilon}} \odot \nabla J(\theta)$$

- Adam (combination of others, β_1, β_2)

test loss
test accuracy

Confusion matrix

		predict		
		a	b	c
Act	a	120	21	7
	b	8	131	63
	c	12	30	80

$$Overall\ Accuracy = \frac{\Sigma diag\ elem}{\#samples}$$
$$Error = 1 - Accuracy$$

Confusion Table

		Predicted	
		Pos	Neg
Act	Pos	TP	FN
	Neg	FP	TN

$$accuracy = \frac{TP + TN}{\#sample}$$
$$sensitivity = recall = \frac{TP}{TP + FN}$$
$$precision = \frac{TP}{TP + FP}$$
$$F\ Score = \frac{2\ prec * reca}{prec + reca}$$
$$specificity = \frac{TN}{TN + FP}$$

Parameter learning
Hyper parameter tuning

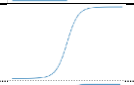
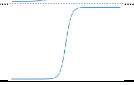
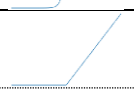
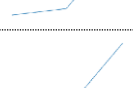
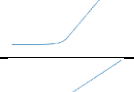
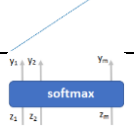
Calculus per layer

one sample	multiple sample
$\boldsymbol{W}^{[l]}: n_l \times n_{l-1}$ $\boldsymbol{b}^{[l]}: n_l \times 1$ $\boldsymbol{a}^{[l-1]}: n_{l-1} \times 1$ $\boldsymbol{a}^{[l]}: n_l \times 1$	$\boldsymbol{W}^{[l]}: n_l \times n_{l-1}$ $\boldsymbol{b}^{[l]}: n_l \times m\ (broadc.)$ $\boldsymbol{A}^{[l-1]}: n_{l-1} \times m$ $\boldsymbol{A}^{[l]}: n_l \times m$
$n_l \times 1$: $\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1]} + \boldsymbol{b}^{[l]}$ $\boldsymbol{a}^{[l]} = g^{[l]}(\boldsymbol{z}^{[l]})$	$n_l \times m$: $\boldsymbol{Z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{A}^{[l-1]} + \boldsymbol{b}^{[l]}$ $\boldsymbol{A}^{[l]} = g^{[l]}(\boldsymbol{Z}^{[l]})$

Chain rule:

$$L\left(g(h(x))\right) \rightarrow dL = \frac{\delta L}{\delta g}\frac{\delta g}{\delta h}\frac{\delta h}{\delta x}dx$$

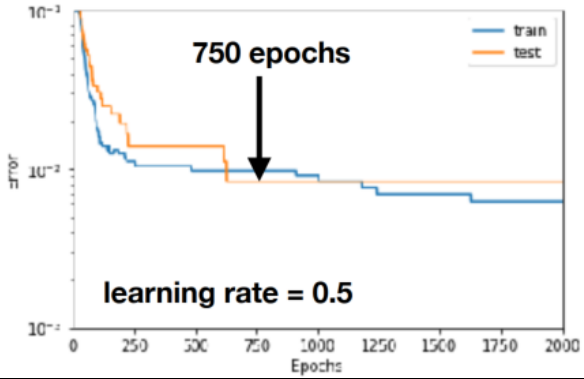
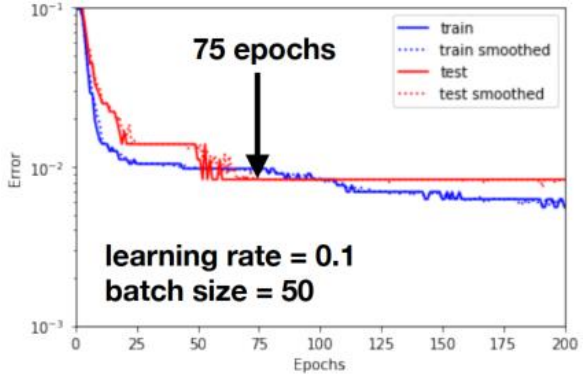
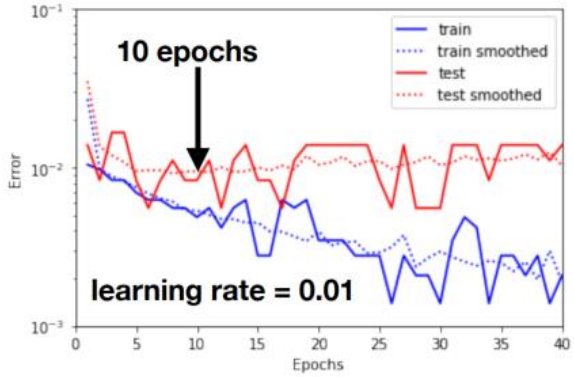
Activation Functions

Name	Output	function	derivate	grafic	good (+)	bad (-)	practical
Heavyside	$[0,1]$	$H(z) = \begin{cases} 1 & (z \geq 0) \\ 0 & (z < 0) \end{cases}$	$H'(z) = 0$		easy	not differentiable	no practical use
Sigmoid	$[0..1]$	$\sigma(z) = \frac{1}{1+e^{-z}}$	$\sigma'(z) = \sigma(z)(1-\sigma(z))$		smooth, gradient works	saturation regions leading to vanishing gradients	illustrative examples, binary classification (output)
Tanh	$[-1..1]$	$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$f'(z) = 1 - \tanh(z)^2$		smooth, gradient works output centred around 0	saturation regions leading to vanishing gradients	better then sigmoid except for output
Recti-Linear unit (ReLU)	$[0..\infty]$	$f(z) = \max(0, z)$	$f'(z) = \begin{cases} 0 & (z < 0) \\ 1 & (z \geq 0) \end{cases}$		Alleviate the vanishing gradient problem	suffer from dying units problem $z < 0$	increasingly used as de facto standard
Leaky Recti-Linear Unit (Leaky ReLU)	$[-\infty..\infty]$	$f(z) = \max(\alpha z, z)$ $0 < \alpha < 1 \rightarrow \alpha = 0.01$	$f'(z) = \begin{cases} a & (z < 0) \\ 1 & (z \geq 0) \end{cases}$		Alleviates both problems (vanishing gradient and dying units)		
Exponential Linear Unit (ELU)	$[-\infty..\infty]$	$f(z) = \begin{cases} \alpha(e^z - 1) & (z < 0) \\ z & (z \geq 0) \end{cases}$				more expensive to compute gradient vanishes at small negative values	
Identity	$[-\infty..\infty]$	$f(z) = z$	$f'(z) = 1$			network is linear -> useless to go deeper (lin+lin=lin)	only used in specific cases
SoftMax		$softmax(\boldsymbol{z}) = \frac{e^z}{\sum_{j=1} e^z}$				used in last layer	multi classification (output)

Layers

Layer	Goal	Parameters	Output shape	# of Parameters, $n_0 = n_x,\ n_L = n_{classes}$
Dense, fully connected	connect two layer	# neurons	$\#\ of\ neurons$	$(n_{l-1} + bias) * n_l$
Dropout	rand disable neurons	p dropout rate	$no\ change$	0
Conv2D	for image processing feature detection	image_size W, # channels C, # filters D, stride s_W, padding P_W, kernel size = filter size w	$\frac{(W - w + 2P_W)}{s_w} + 1 \quad P_W = \frac{w-1}{2}$	$D * \left(\underbrace{whC}_{weight} + \underbrace{1}_{bias} \right)$
Activation (e.g. ReLU)	insert non-linearity	only in Leaky-ReLU or ELU: α	$no\ change$	0
MaxPooling2D	reduce output size	pool size	$\frac{input\ size}{pool\ size}$	0
Flatten	to calculate properly	-	$width * height * depth$	0
SimpleRNN	for sequential data	# units, # unique_chars (=features)	$\#\ of\ units$	$(\#\ \text{features} + \#\ units + 1) * units$
Batch-Normalis. / Gradient clip	faster learning		$no\ change$	
Fall Back layers	for deep networks			
Bottleneck layer	reduce params			
GRU (Gated Recurrent Units)				
LSTM (Long Short Term Memory)	longer memory			

Learning Schemes

BGD: Batch Gradient Descent	**MBGD: Mini-Batch Gradient Descent** (Compromise)	**SGD: Stochastic Gradient Descent**
		
Summation over all the training set.	Summation only over subset of the training samples.	No summation, just a single sample.
Smooth	Wiggling curves, need smoothing.	Slightly wiggling curves.
Strictly decreasing curves	Cost curve not necessarily decreasing. Ends up wandering closely around the global minimum - not a problem in practise.	
Slowest, update for all training samples.	Faster for large m. Much less epochs than BGD, but more than SGD needed. Faster since parameters are updated for each mini-batch	Faster since parameters are updates for each training sample.
All data in RAM (No **out-of-core support**)	Can handle very large data sets, that do not fit in memory (**out-of-core learning**)	
No incremental learning possible.	Allows incremental learning (**online learning, on-the-fly**)	
Easily parallelised on GPU and HPC.		Not easily parallelised.
The learning principle is generalisable to many other "hypothesis families"		

`for each epoch:`
$\Delta w_j \coloneqq \alpha * \sum_i (target^{(i)} - output^{(i)}) x_j^{(i)}$
`for each weight j:`
$w_j \leftarrow w_j - \Delta w_j$
	`for each epoch:`
	$\Delta w_j = \alpha * \Sigma_i^b (target^{(i)} - output^{(i)}) x_j^{(i)}$
	`for each minibatch:`
	`for each weight j:`
	$w_j \leftarrow w_j - \Delta w_j$
		`for each epoch:`
		`i := rand training sample:`
		$\Delta w_j = \alpha * (target^{(i)} - output^{(i)}) x_j^{(i)}$
		`for each weight j:`
		$w_j \leftarrow w_j - \Delta w_j$

1) Start with some initial value for the parameter vector θ_0
2) Iteratively update the parameter vector by
a. Computing the gradient of the cost function $J(\theta)$ at the last position reached (θ_t): $\nabla_\theta J(\theta_t)$
b. Stepping in the negative direction of the gradient according to
$\theta_{t+1} = \theta_t - \alpha * \nabla_\theta J(\theta_t)$
3) Stop when change in parameter vector small
	`W,b = initialize_weights(nx, ny)`
	`X = shuffle(X)`
	`foreach(epoch in epochs)`
	`minibatches = get_next_minibatch(X)`
	`foreach(minibatch in minibatches)`
	$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha(h_\theta(\boldsymbol{x}^{(i)}) - y^{(i)})\,\boldsymbol{x}^{(i)}$
	$b \leftarrow b - \alpha(h_\theta(\boldsymbol{x}^{(i)}) - y^{(i)})$
		`W,b = initialize_weights(nx, ny)`
		`X = shuffle(X)`
		`foreach(epoch in epochs)`
		`foreach(x in X)`
		$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha(h_\theta(\boldsymbol{x}^{(i)}) - y^{(i)})\,\boldsymbol{x}^{(i)}$
		$b \leftarrow b - \alpha(h_\theta(\boldsymbol{x}^{(i)}) - y^{(i)})$