CS480/680: Introduction to Machine Learning

Lec 08: Multilayer Perceptron

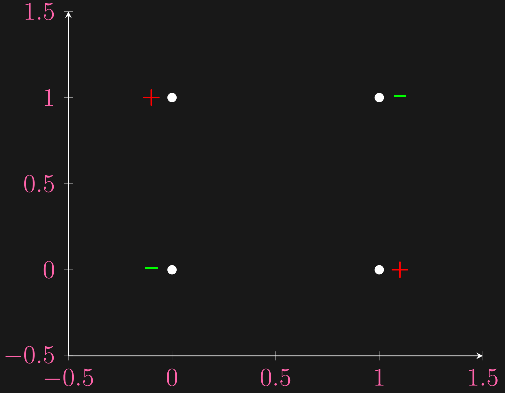
Yaoliang Yu

UNIVERSITY OF WATERLOO | FACULTY OF MATHEMATICS
DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE

June 03, 2024

XOR recalled

	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$
	0	1	0	1
	0	0	1	1
$\mathbf{y}$	−	+	+	−



No Separating Hyperplane

$$\mathsf{y}(\langle \mathbf{x}, \mathbf{w} \rangle + b) > 0$$

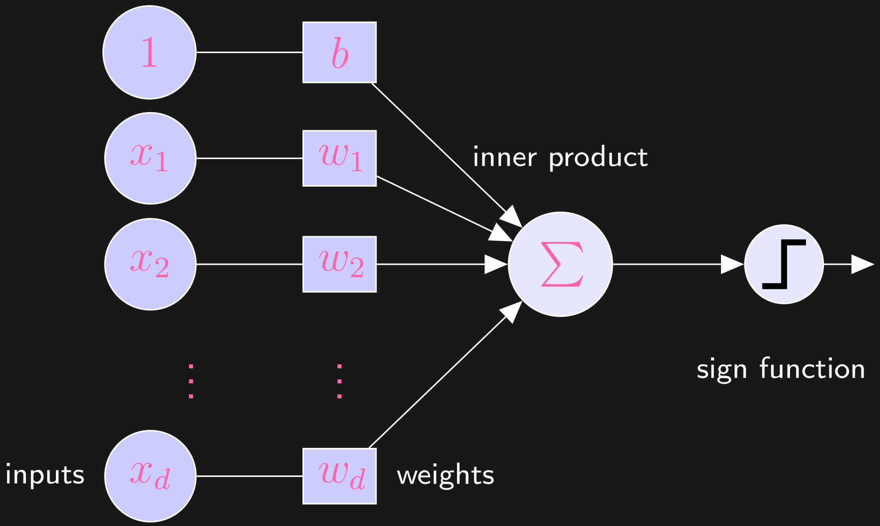
- $\mathbf{x}_1 = (0,0), \mathsf{y}_1 = - \implies b < 0$
- $\mathbf{x}_2 = (1,0), \mathsf{y}_1 = + \implies w_1 + b > 0$
- $\mathbf{x}_3 = (0,1), \mathsf{y}_1 = + \implies w_2 + b > 0$
- $\mathbf{x}_4 = (1,1), \mathsf{y}_1 = - \implies w_1 + w_2 + b < 0$

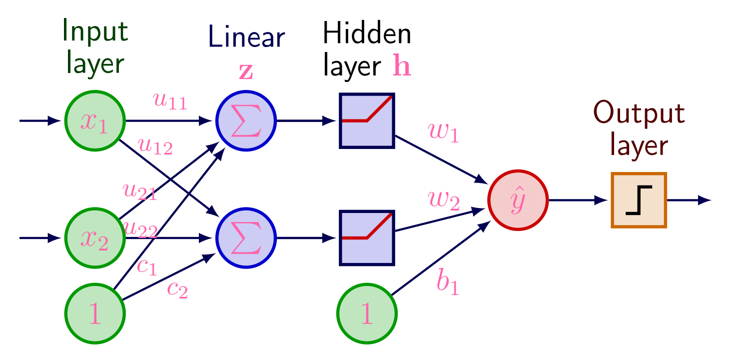
$\boxed{\text{Contradiction!}}$

Fixing the Problem

- Our model (a hyperplane in the input space) underfits the data (generated by a complicated function such as XOR)
- Can fix input representation but use a richer model (e.g. an ellipsoid)
- Can fix hyperplane as classifier but use a richer input representation
- The two approaches are equivalent, through a reproducing kernel
- Neural network: learn the feature map simultaneously with the hyperplane!

From 1 to 2


1
b
x_1
w_1
inner product
x_2
w_2
Σ
sign function
x_d
w_d
inputs
weights

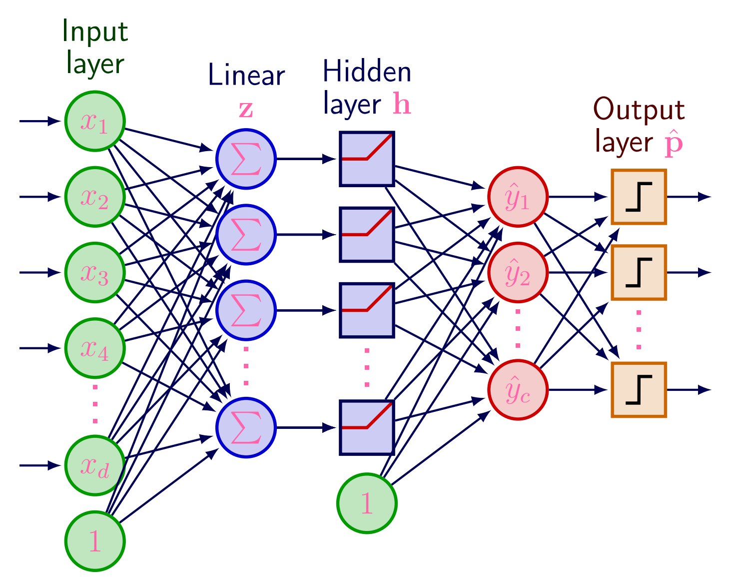


- 1st linear transformation: $\mathbf{z} = U\mathbf{x} + \mathbf{c}, \quad U \in \mathbb{R}^{2\times 2}, \ \mathbf{c} \in \mathbb{R}^2$
- Element-wise nonlinear activation: $\mathbf{h} = \sigma(\mathbf{z})$; makes all the difference!
- 2nd linear transformation: $\hat{y} = \langle \mathbf{h}, \mathbf{w} \rangle + b$

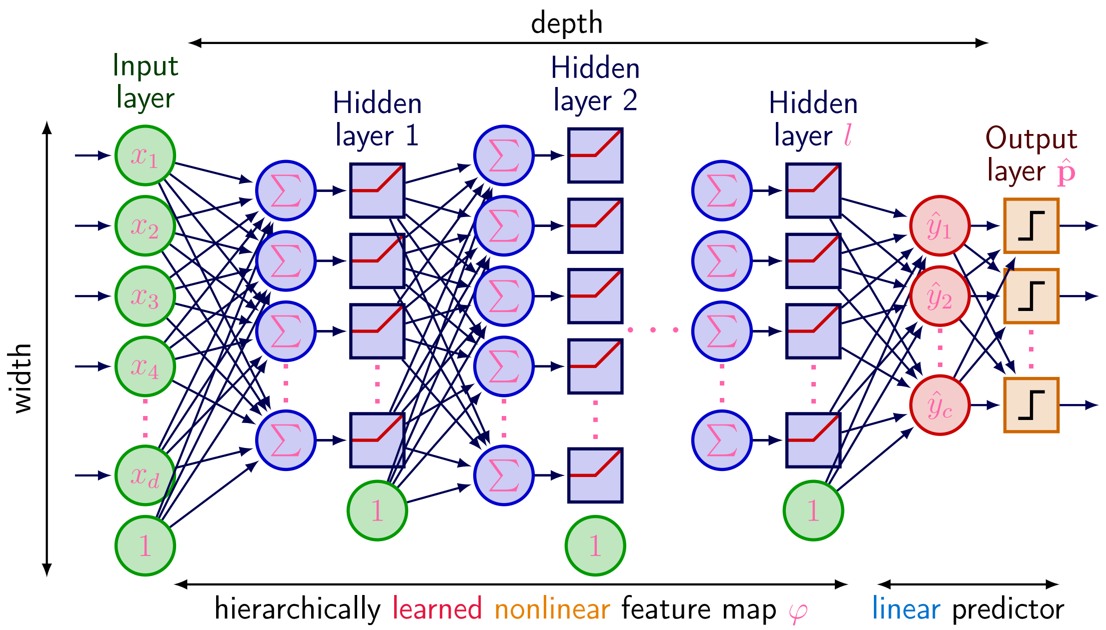
Does It Work?

$$U = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 2 \\ -4 \end{bmatrix}, b = -1$$

- Rectified Linear Unit (ReLU): $\sigma(t) = t_+ := \max\{t, 0\}$
- $\mathbf{x}_1 = (0, 0), \mathrm{y} = - \implies \mathbf{z}_1 = (0, -1), \mathbf{h}_1 = (0, 0) \implies \hat{y} = -1$ ✅
- $\mathbf{x}_2 = (1, 0), \mathrm{y} = + \implies \mathbf{z}_2 = (1, \quad 0), \mathbf{h}_2 = (1, 0) \implies \hat{y} = +1$ ✅
- $\mathbf{x}_3 = (0, 1), \mathrm{y} = + \implies \mathbf{z}_3 = (1, \quad 0), \mathbf{h}_3 = (1, 0) \implies \hat{y} = +1$ ✅
- $\mathbf{x}_4 = (1, 1), \mathrm{y} = - \implies \mathbf{z}_4 = (2, \quad 1), \mathbf{h}_4 = (2, 1) \implies \hat{y} = -1$ ✅



$$\mathbf{z} = U\mathbf{x} + \mathbf{c}, \quad \mathbf{h} = \sigma(\mathbf{z}), \quad \hat{\mathbf{y}} = W\mathbf{h} + \mathbf{b}, \quad \hat{\mathbf{p}} = \mathtt{softmax}(\hat{\mathbf{y}})$$


depth
Input layer
Hidden layer 1
Hidden layer 2
Hidden layer l
Output layer $\hat{\mathbf{p}}$
width
x_1
x_2
x_3
x_4
x_d
1
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_c$
hierarchically learned nonlinear feature map φ
linear predictor

Algorithm: Feed-forward MLP during test time

Input: $\mathbf{x} \in \mathbb{R}^{d_0}$, activation function $\sigma : \mathbb{R} \to \mathbb{R}$

Output: $\hat{\mathbf{p}} \in \mathbb{R}^{c}$

$\mathbf{h}_0 \leftarrow \mathbf{x}$ // initialize with input data

for $k = 1, \ldots, l$ **do** // feature map: layer by layer

  $\mathbf{z}_k \leftarrow W_k \mathbf{h}_{k-1} + \mathbf{b}_k$ // $W_k \in \mathbb{R}^{d_k \times d_{k-1}}, \mathbf{b}_k \in \mathbb{R}^{d_k}$

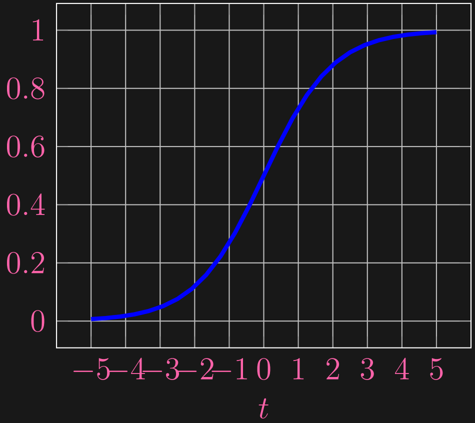
  $\mathbf{h}_k \leftarrow \sigma(\mathbf{z}_k)$ // element-wise

$\hat{\mathbf{y}} \leftarrow W_{l+1} \mathbf{h}_l + \mathbf{b}_{l+1}$ // $W_{l+1} \in \mathbb{R}^{c \times d_l}, \mathbf{b}_{l+1} \in \mathbb{R}^{c}$

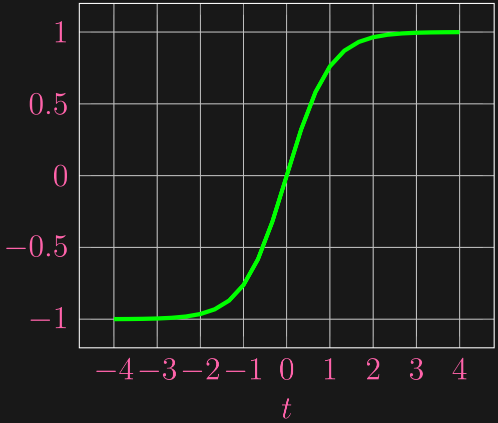
$\hat{\mathbf{p}} \leftarrow \mathtt{softmax}(\hat{\mathbf{y}})$ // $\mathtt{softmax}(\mathbf{a}) = \exp(\mathbf{a}) / \langle \exp(\mathbf{a}), \mathbf{1} \rangle$

Activation Function

$$\mathrm{sgm}(t) = \frac{1}{1+\exp(-t)}$$

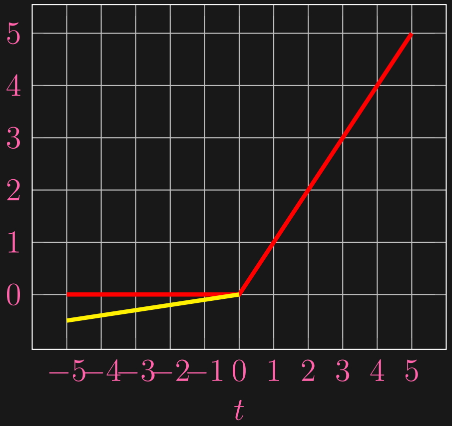


$$\tanh(t) = 1 - 2\mathrm{sgm}(2t)$$

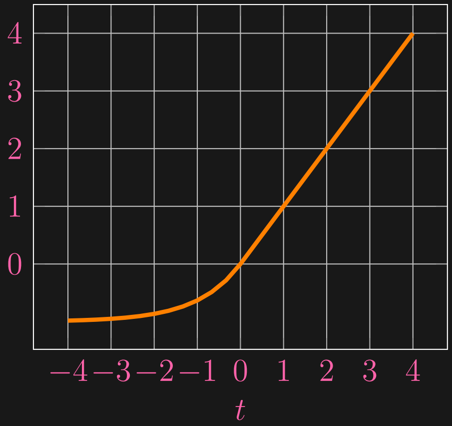


Activation Function cont'

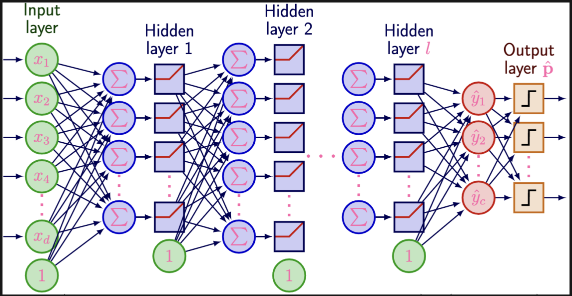
$\texttt{relu}(t) = t_+$



$\texttt{elu}(t) = (t)_+ + (t)_-(\exp(t) - 1)$



MLP Training



$$\hat{\mathbf{p}} = f(\mathbf{x}; \mathbf{w})$$

- Need a loss ℓ to measure difference between prediction $\hat{\mathbf{p}}$ and truth y
  - e.g. squared loss $\|\hat{\mathbf{p}} - \mathsf{y}\|_2^2$ or log-loss $-\log \hat{p}_{\mathsf{y}}$
- Need a training set $\mathcal{D} = \{(\mathbf{x}_i, \mathsf{y}_i) : i = 1, \ldots, n\}$ to train weights $\mathbf{w}$

Stochastic Gradient (SGD)

$$\min_{\mathbf{w}} \frac{1}{n}\sum_{i=1}^{n}[\ell \circ f](\mathbf{x}_i, \mathsf{y}_i; \mathbf{w})$$

$$\boxed{\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \frac{1}{n}\sum_{i=1}^{n}\nabla[\ell \circ f](\mathbf{x}_i, \mathsf{y}_i; \mathbf{w})}$$

- $[\ell \circ f](\mathbf{x}_i, \mathsf{y}_i; \mathbf{w}) := \ell[f(\mathbf{x}_i; \mathbf{w}), \mathsf{y}_i]$
- Each iteration requires a full pass over the entire training set!
- A random, minibatch $B \subseteq \{1, \ldots, n\}$ suffices:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \frac{1}{|B|}\sum_{i \in B}\nabla[\ell \circ f](\mathbf{x}_i, \mathsf{y}_i; \mathbf{w})$$

- Trade-off between variance and computation

Momentum

$$\mathbf{w}_{t+1} = \underbrace{\mathbf{w}_t - \eta_t \nabla g(\mathbf{w}_t)}_{\text{gradient step}} + \underbrace{\beta_t(\mathbf{w}_t - \mathbf{w}_{t-1})}_{\text{momentum}} = \underbrace{(1+\beta_t)\mathbf{w}_t - \beta_t \mathbf{w}_{t-1}}_{\text{extrapolation}} - \eta_t \nabla g(\mathbf{w}_t)$$

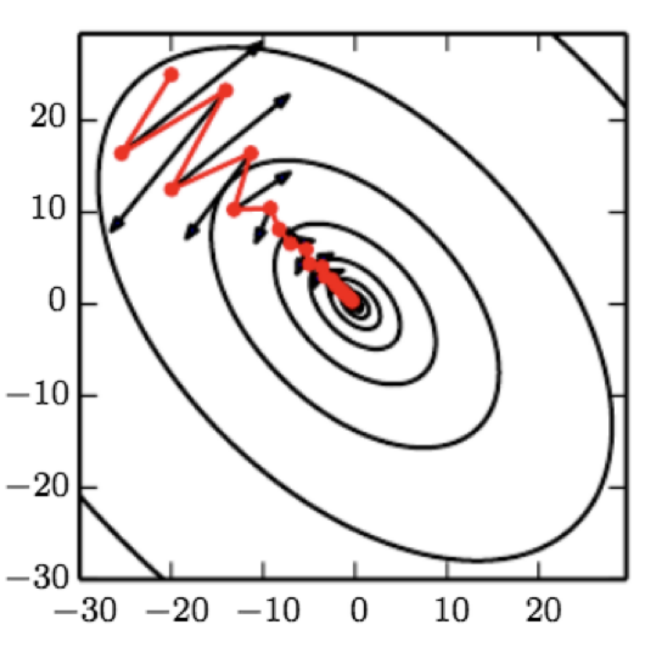
Using the intermediate $\tilde{\mathbf{v}}$ defined below:

$$\tilde{\mathbf{v}}_{t+1} = \beta_t \tilde{\mathbf{v}}_t + \eta_t \nabla g(\mathbf{w}_t), \quad \text{where} \quad \tilde{\mathbf{v}}_{t+1} = \mathbf{w}_t - \mathbf{w}_{t+1}$$

We can also interpret momentum as gradient averaging:

$$\begin{aligned}\mathbf{v}_{t+1} &= \tfrac{\alpha_{t-1}}{\alpha_t}\beta_t \mathbf{v}_t + \tfrac{\eta_t}{\alpha_t}\nabla g(\mathbf{w}_t), \quad \text{where} \quad \alpha_t \mathbf{v}_{t+1} = \mathbf{w}_t - \mathbf{w}_{t+1} \\ &= \underbrace{(1-\gamma_t)\mathbf{v}_t + \gamma_t \nabla g(\mathbf{w}_t)}_{\text{gradient averaging}}, \quad \underbrace{\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha_t \mathbf{v}_{t+1}}_{\text{gradient update}},\end{aligned}$$

$$\text{where} \quad \alpha_t = \alpha_{t-1}\beta_t + \eta_t \quad \text{and} \quad \gamma_t := \tfrac{\eta_t}{\alpha_t} \in [0,1]$$


20
10
0
−10
−20
−30
−30
−20
−10
0
10
20

Nesterov Momentum

$$\mathbf{w}_{t+1} = \mathbf{z}_t - \eta_t \nabla g(\mathbf{z}_t), \qquad \mathbf{z}_{t+1} = \mathbf{w}_{t+1} + \beta_t(\mathbf{w}_{t+1} - \mathbf{w}_t)$$

Using the intermediate steps below:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla g(\mathbf{z}_t) + \beta_{t-1}(\mathbf{w}_t - \mathbf{w}_{t-1}), \qquad \mathbf{z}_{t+1} = \mathbf{w}_{t+1} + \beta_t(\mathbf{w}_{t+1} - \mathbf{w}_t)$$

$$\tilde{\mathbf{v}}_{t+1} = \beta_{t-1}\tilde{\mathbf{v}}_t + \eta_t \nabla g(\mathbf{z}_t), \quad \text{where} \quad \tilde{\mathbf{v}}_{t+1} = \mathbf{w}_t - \mathbf{w}_{t+1}, \quad \mathbf{z}_t = \mathbf{w}_t - \beta_{t-1}\tilde{\mathbf{v}}_t$$

We can again interpret Nesterov momentum as averaging gradients looked ahead:

$$\begin{aligned}
\mathbf{v}_{t+1} &= \tfrac{\alpha_{t-1}}{\alpha_t}\beta_{t-1}\mathbf{v}_t + \tfrac{\eta_t}{\alpha_t}\nabla g(\mathbf{z}_t), \quad \text{where} \quad \alpha_t \mathbf{v}_{t+1} = \mathbf{w}_t - \mathbf{w}_{t+1} \\
&= \underbrace{(1-\gamma_t)\mathbf{v}_t + \gamma_t \nabla g(\mathbf{z}_t),}_{\text{gradient averaging}} \quad \underbrace{\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha_t \mathbf{v}_{t+1},}_{\text{gradient update}}
\end{aligned}$$

$$\text{where} \quad \mathbf{z}_t = \mathbf{w}_t - \beta_{t-1}\alpha_{t-1}\mathbf{v}_t = \mathbf{w}_t - \alpha_t(1-\gamma_t)\mathbf{v}_t \quad \text{looks ahead}$$

$$\alpha_t = \alpha_{t-1}\beta_{t-1} + \eta_t, \;\; \gamma_t := \tfrac{\eta_t}{\alpha_t} \in [0,1]$$

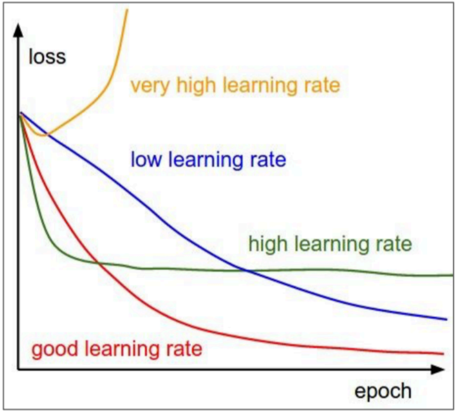
Adagrad and Adam

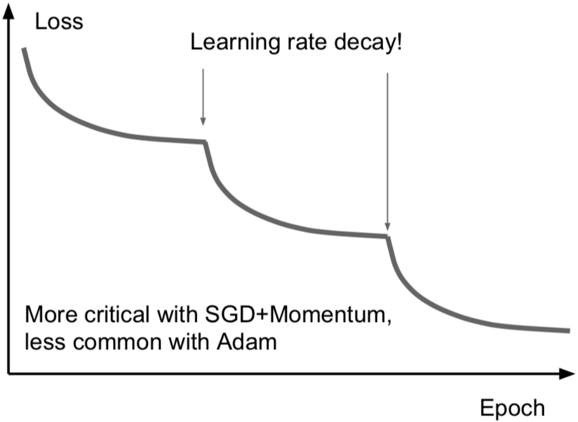
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta_t}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{v}_t$$

	$\mathbf{s}_t$	$\mathbf{v}_t$
Adagrad	$\mathbf{s}_{t-1} + \nabla g(\mathbf{w}_t) \odot \nabla g(\mathbf{w}_t)$	$\nabla g(\mathbf{w}_t)$
RMSprop	$(1-\lambda_t)\mathbf{s}_{t-1} + \lambda_t \nabla g(\mathbf{w}_t) \odot \nabla g(\mathbf{w}_t)$	$\nabla g(\mathbf{w}_t)$
Adam	$(1-\lambda_t)\mathbf{s}_{t-1} + \lambda_t \nabla g(\mathbf{w}_t) \odot \nabla g(\mathbf{w}_t)$	$(1-\gamma_t)\mathbf{v}_{t-1} + \gamma_t \nabla g(\mathbf{w}_t)$

bias correction: divide by $\sum_{k=1}^{t} \lambda_k \prod_{l>k}(1-\lambda_l)$, similar for γ_t [to retain constant grad]

J. Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". *Journal of Machine Learning Research*, vol. 12, no. 61 (2011), pp. 2121–2159, D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations*. 2015.





- Decrease every few epochs: $\eta_t = \eta_0 \exp(-c \lceil t/k \rceil)$
- Sublinear decay: $\eta_t = \eta_0/(1 + ct)$ or $\eta_t = \eta_0/\sqrt{1 + ct}$

Computational Graph

- A DAG (directed acyclic graph) $(\mathscr{V}, \mathscr{E})$ with 3 kinds of nodes:

$$\underbrace{v_1, \ldots, v_d,}_{\text{input}} \quad \underbrace{v_{d+1}, \ldots, v_{d+k},}_{\text{intermediate variables}} \quad \underbrace{v_{d+k+1}, \ldots, v_{d+k+m}}_{\text{output}}$$

- Arranged in an order so that $(v_i, v_j) \in \mathscr{E} \implies i < j$
- For each v_i, define $\mathscr{I}_i := \{u \in \mathscr{V} : (u, v_i) \in \mathscr{E}\}$ and $\mathscr{O}_i := \{u \in \mathscr{V} : (v_i, u) \in \mathscr{E}\}$
- Sequentially for each $i = 1, \ldots, d + k + m$, we compute

$$v_i = \begin{cases} w_i, & i \le d \\ f_i(\mathscr{I}_i), & i > d \end{cases}$$

F. L. Bauer. "Computational Graphs and Rounding Error". *SIAM Journal on Numerical Analysis*, vol. 11, no. 1 (1974), pp. 87–96,
L. V. Kantorovich. "On a system of mathematical symbols, convenient for electronic computer operations". *Soviet Mathematics Doklady*, vol. 113, no. 4 (1957), pp. 738–741.

How to Compute the Gradient?

- Forward differentiation: let $U_i := \frac{\partial v_i}{\partial \mathbf{w}} \in \mathbb{R}^{d \times d_i}$, then

$$U_i = \sum_{j \in \mathscr{I}_i} U_j \cdot \nabla_j f_i, \quad \text{where} \quad \nabla_j f_i = \frac{\partial f_i}{\partial v_j} \in \mathbb{R}^{d_j \times d_i}$$

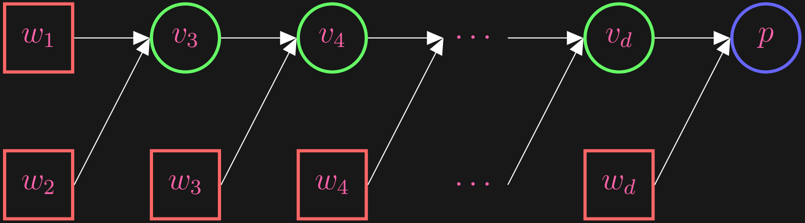
- Backward differentiation: let $V_i := \frac{\partial \mathbf{p}}{\partial v_i} \in \mathbb{R}^{d_i \times m}$, then

$$V_i = \sum_{j \in \mathscr{O}_i} \nabla_i f_j \cdot V_j, \quad \text{where} \quad \nabla_i f_j = \frac{\partial f_j}{\partial v_i} \in \mathbb{R}^{d_i \times d_j}$$

- When to use which?

Speelpenning's Example

$$p = f(\mathbf{w}) = \prod_j w_j, \quad \frac{\mathrm{d}p}{\mathrm{d}\mathbf{w}} = \left[\prod_{j\neq 1} w_j, \ \prod_{j\neq 2} w_j, \ \cdots \ \prod_{j\neq d} w_j\right]$$



B. Speelpenning. "Compiling Fast Partial Derivatives of Functions Given by Algorithms". PhD thesis. University of Illinois at Urbana-Champaign, 1980.

Forward-mode: for $j = 2, 3, \ldots$, $v_2 := w_1$ and $\frac{\mathrm{d}v_2}{\mathrm{d}\mathbf{w}} := \mathbf{e}_1$,

$$\boxed{v_{j+1} = v_j \cdot w_j}$$

$$\frac{\mathrm{d}v_{j+1}}{\mathrm{d}\mathbf{w}} = \frac{\mathrm{d}v_j}{\mathrm{d}\mathbf{w}} \cdot w_j + v_j \cdot \mathbf{e}_j,$$

which costs $O(j)$ as we may ignore the 0s and update only the first j entries.

Reverse-mode: for $j = d, d-1, \ldots$, $v_{d+1} := p$ and $\frac{\mathrm{d}p}{\mathrm{d}v_{d+1}} := 1$,

$$\frac{\mathrm{d}p}{\mathrm{d}v_j} = \frac{\mathrm{d}p}{\mathrm{d}v_{j+1}} w_j, \quad \frac{\mathrm{d}p}{\mathrm{d}w_j} = \frac{\mathrm{d}p}{\mathrm{d}v_{j+1}} v_j \implies \frac{\mathrm{d}p}{\mathrm{d}v_j} = \prod_{k=j}^{d} w_k, \quad v_j = \prod_{k=1}^{j-1} w_k, \quad \frac{\mathrm{d}p}{\mathrm{d}w_j} = \prod_{k \neq j} w_k.$$

Algorithm: Feed-forward MLP trained with backpropogation

Input: $\mathbf{x} \in \mathbb{R}^{d_0}$, activation $\sigma : \mathbb{R} \to \mathbb{R}$, loss $\ell : \mathbb{R}^c \to \mathbb{R}$, regularizer r

$\mathbf{h}_0 \leftarrow \mathbf{x}$ `// initialize with input data`

for $k = 1, \ldots, l$ **do** `// feature map: layer by layer`

  $\mathbf{z}_k \leftarrow W_k \mathbf{h}_{k-1} + \mathbf{b}_k$ // $W_k \in \mathbb{R}^{d_k \times d_{k-1}}, \mathbf{b}_k \in \mathbb{R}^{d_k}$

  $\mathbf{h}_k \leftarrow \sigma(\mathbf{z}_k)$ `// element-wise`

$\hat{\mathbf{y}} \leftarrow W_{l+1} \mathbf{h}_l + \mathbf{b}_{l+1}$ // $W_{l+1} \in \mathbb{R}^{c \times d_l}, \mathbf{b}_{l+1} \in \mathbb{R}^c$

$\hat{\mathbf{p}} \leftarrow \mathtt{softmax}(\hat{\mathbf{y}})$ // $\mathtt{softmax}(\mathbf{a}) = \exp(\mathbf{a}) / \langle \exp(\mathbf{a}), \mathbf{1} \rangle$

$\frac{\partial \ell}{\partial W_{l+1}} \leftarrow \frac{\partial \hat{\mathbf{y}}}{\partial W_{l+1}} \cdot \frac{\partial \hat{\mathbf{p}}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \ell}{\partial \hat{\mathbf{p}}} + \frac{\partial r}{\partial W_{l+1}}$ // we absorb $\mathbf{b}_k$ into W_k

$\frac{\partial \ell}{\partial \mathbf{h}_l} \leftarrow \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{h}_l} \cdot \frac{\partial \hat{\mathbf{p}}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \ell}{\partial \hat{\mathbf{p}}}$ `// initialize`

for $k = l, \ldots, 1$ **do** `// backward: accumulate derivatives`

  $\frac{\partial \ell}{\partial \mathbf{z}_k} \leftarrow \frac{\partial \mathbf{h}_k}{\partial \mathbf{z}_k} \cdot \frac{\partial \ell}{\partial \mathbf{h}_k}$ // $\frac{\partial \mathbf{h}_k}{\partial \mathbf{z}_k} = \mathrm{diag}(\sigma'(\mathbf{z}_k))$

  $\frac{\partial \ell}{\partial W_k} \leftarrow \frac{\partial \mathbf{z}_k}{\partial W_k} \cdot \frac{\partial \ell}{\partial \mathbf{z}_k} + \frac{\partial r}{\partial W_k}$ // $\frac{\partial \mathbf{z}_k}{\partial W_k} = \sum_j [\mathbf{h}_{k-1}^\top \otimes \mathbf{e}_j] \otimes \mathbf{e}_j$

  $\frac{\partial \ell}{\partial \mathbf{h}_{k-1}} \leftarrow \frac{\partial \mathbf{z}_k}{\partial \mathbf{h}_{k-1}} \cdot \frac{\partial \ell}{\partial \mathbf{z}_k}$ // $\frac{\partial \mathbf{z}_k}{\partial \mathbf{h}_{k-1}} = W_k^\top$

Universal Representation

Theorem: Addition is the only continuous multi-variate function

For any $d \in \mathbb{N}$ there exist constants $\lambda_j > 0, j = 1, \ldots, d, \mathbf{1}^\top \boldsymbol{\lambda} < 1$ and strictly increasing Lipschitz continuous functions $\varphi_k : [0,1] \to [0,1], k = 1, \ldots, 2d+1$ such that for any continuous function $f : [0,1]^d \to \mathbb{R}$ there exists some continuous univariate function $\sigma : [0,1] \to \mathbb{R}$ so that

$$f(\mathbf{x}) = \sum_{k=1}^{2d+1} \sigma\left(\sum_{j=1}^{d} \lambda_j \varphi_k(x_j)\right)$$

- Try $f(x_1, x_2) = x_1 x_2$?

J.-P. Kahane. "Sur le théorème de superposition de Kolmogorov". *Journal of Approximation Theory*, vol. 13, no. 3 (1975), pp. 229–234, A. Kolmogorov. "The theory of transmission of information". In: *The Session of the Academy of Sciences of the USSR on Scientific Problems of Industrial Automation*. 1957, pp. 66–99, V. I. Arnol'd. "On Functions of Three Variables". *Soviet Mathematics Doklady*, vol. 114, no. 4 (1957), pp. 679–681.

Theorem: Universal approximation

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be Riemann integrable on any bounded interval. Then, two-layer NN with activation function σ is "dense" in $\mathcal{C}(\mathbb{R}^d)$ iff σ is not a polynomial (a.e.).

- The set of two-layer NN with activation function σ:

$$\operatorname{span}\{\mathbf{x} \mapsto \sigma(\langle \mathbf{x}, \mathbf{w} \rangle + b) : \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}\}$$

- Lebesgue proved that a function is Riemann integrable (over a bounded interval) iff it is bounded and continuous almost everywhere (a.e.)

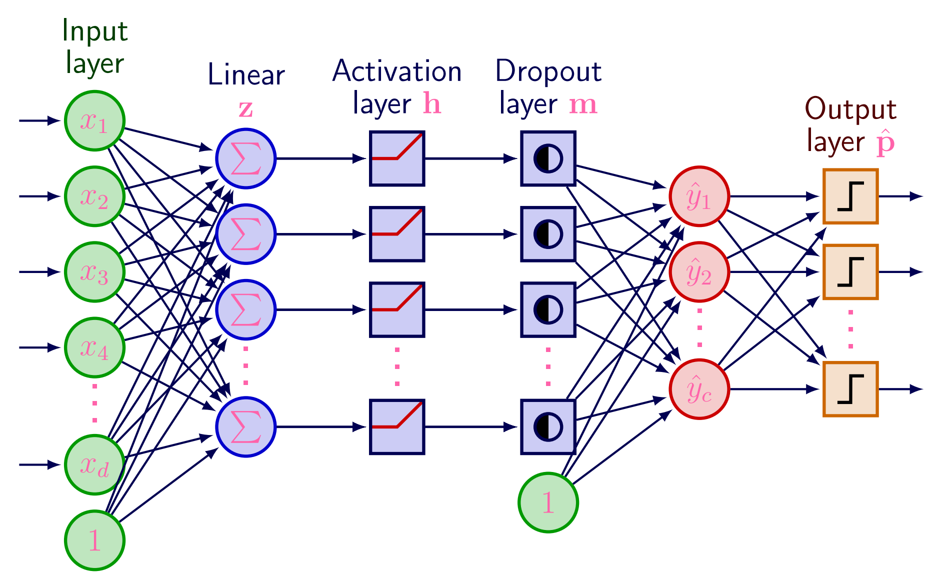
- The indicator function $x \mapsto [\![x \in \mathbb{Q}]\!]$ over rationals $\mathbb{Q}$ is a counterexample

M. Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". *Neural Networks*, vol. 6, no. 6 (1993), pp. 861–867, K. Hornik. "Some new results on neural network approximation". *Neural Networks*, vol. 6, no. 8 (1993), pp. 1069–1072.

Dropout

- For each training minibatch, keep each hidden unit with probability q
- A different and random network for each training minibatch
- A smaller network with less capacity
- Hidden units are less likely to collude to overfit training data
- Inverted: multiplying each $\mathbf{h}_k$ with a scaled binary mask $\mathbf{m}/q$
- Use the full network for testing

N. Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". *Journal of Machine Learning Research*, vol. 15, no. 56 (2014), pp. 1929–1958.



Input layer
Linear z
Activation layer h
Dropout layer m
Output layer $\hat{\mathbf{p}}$
x_1
x_2
x_3
x_4
x_d
1
$\sum$
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_c$
1

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1 \dots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

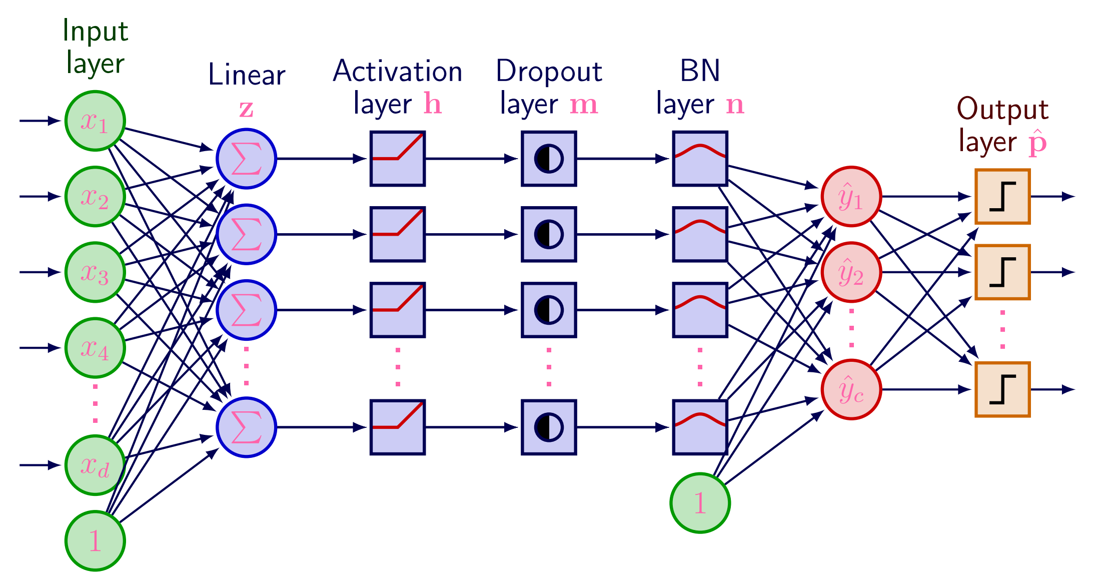
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

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

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

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

S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. vol. 37. 2015, pp. 448–456.



Input layer
Linear $\mathbf{z}$
Activation layer $\mathbf{h}$
Dropout layer $\mathbf{m}$
BN layer $\mathbf{n}$
Output layer $\hat{\mathbf{p}}$
x_1
x_2
x_3
x_4
x_d
1
Σ
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_c$
1