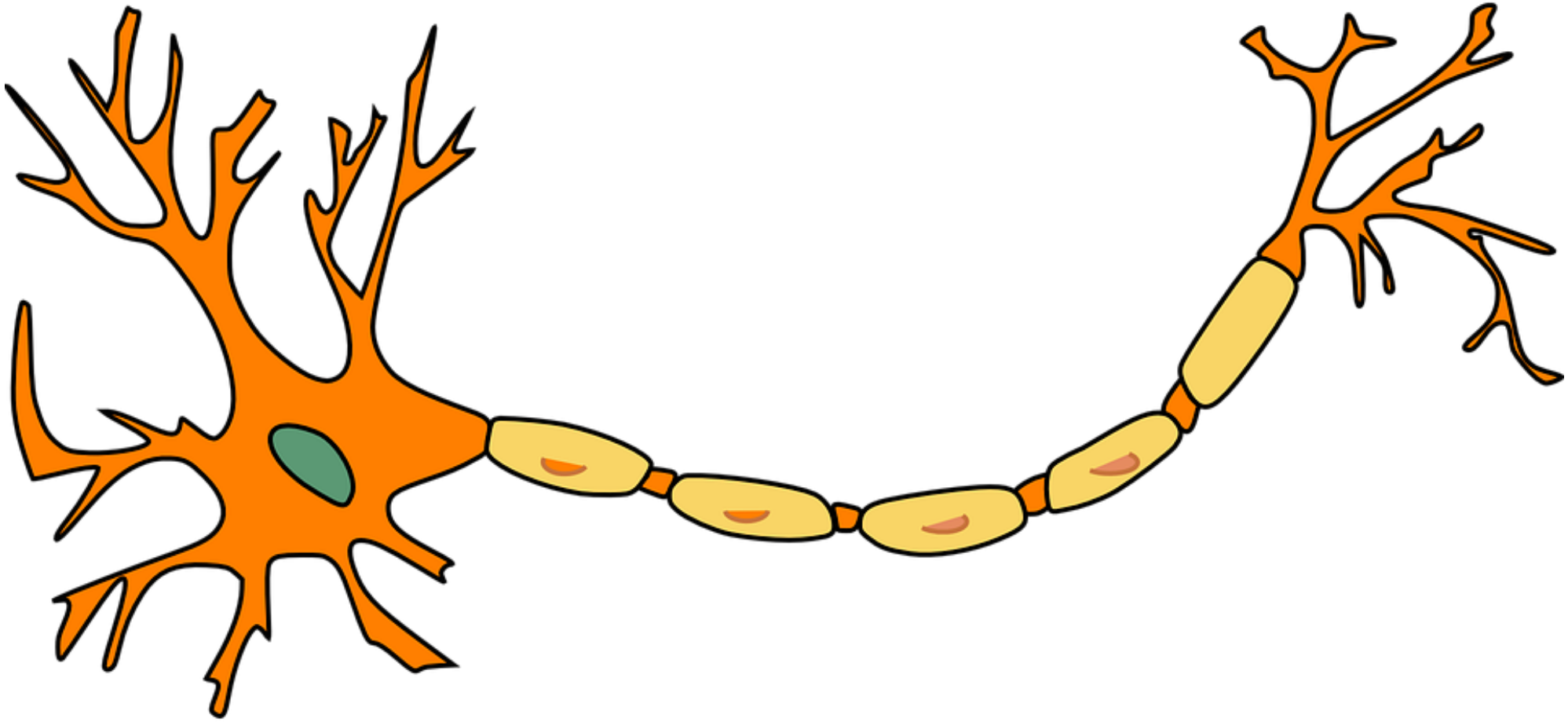
CS 594 Modern Reinforcement Learning

Lecture 6: Function Approximation

Neurons



Invention of artificial neural networks*

The bulletin of mathematical biophysics
December 1943, Volume 5, Issue 4, pp 115–133 | Cite as

A logical calculus of the ideas immanent in nervous activity

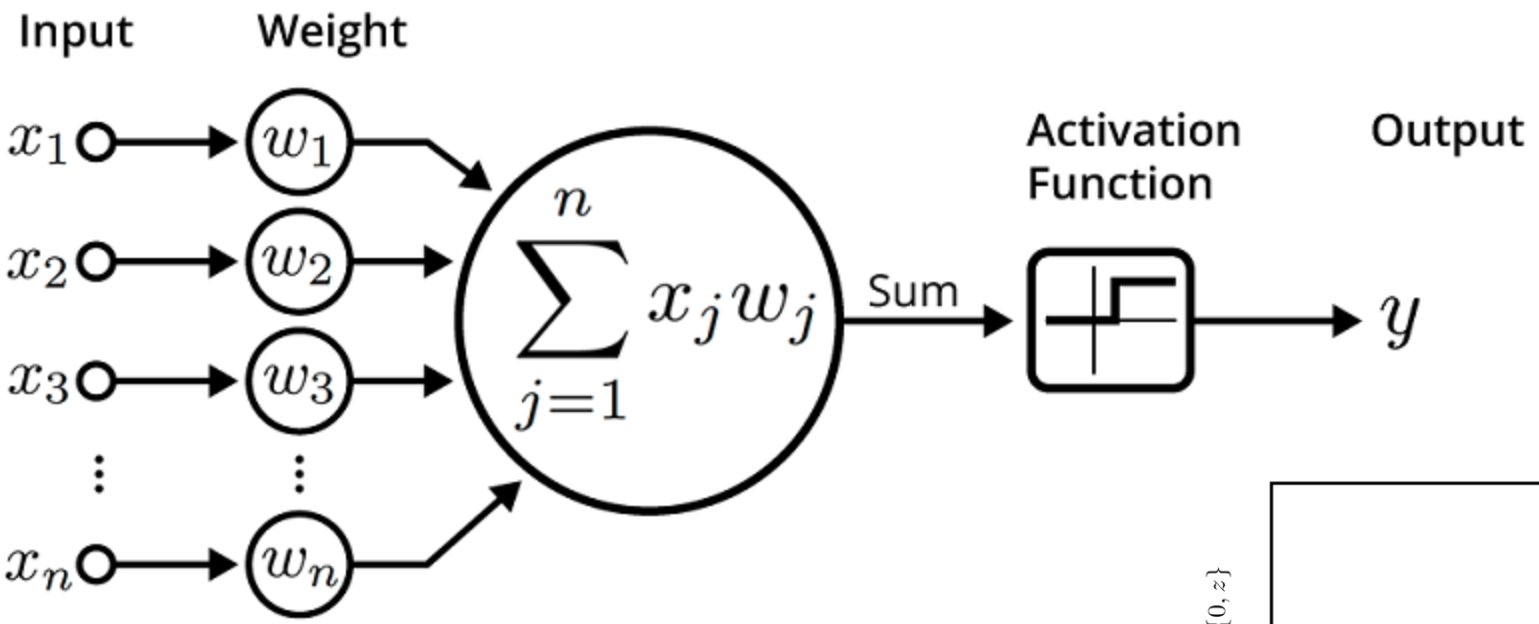
Authors Authors and affiliations

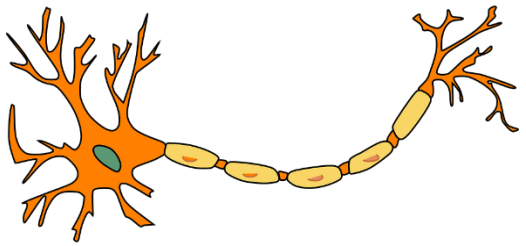
Warren S. McCulloch, Walter Pitts

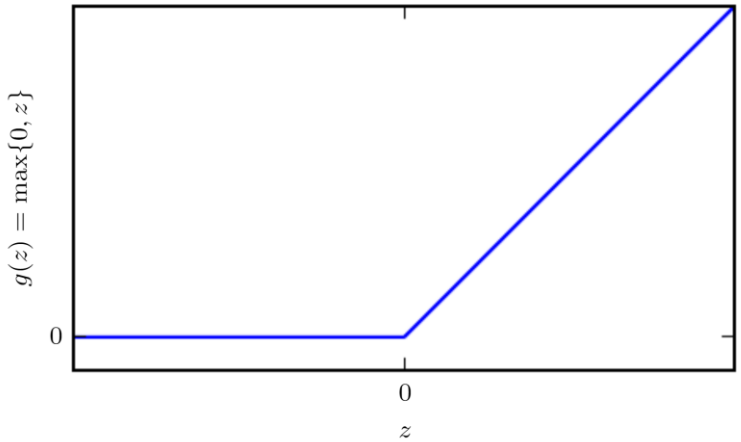
"In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons might work … modeled a simple neural network using electrical circuits." https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

Warren McCulloch UIC College of Medicine; Walter Pitts University of Chicago

Neurons

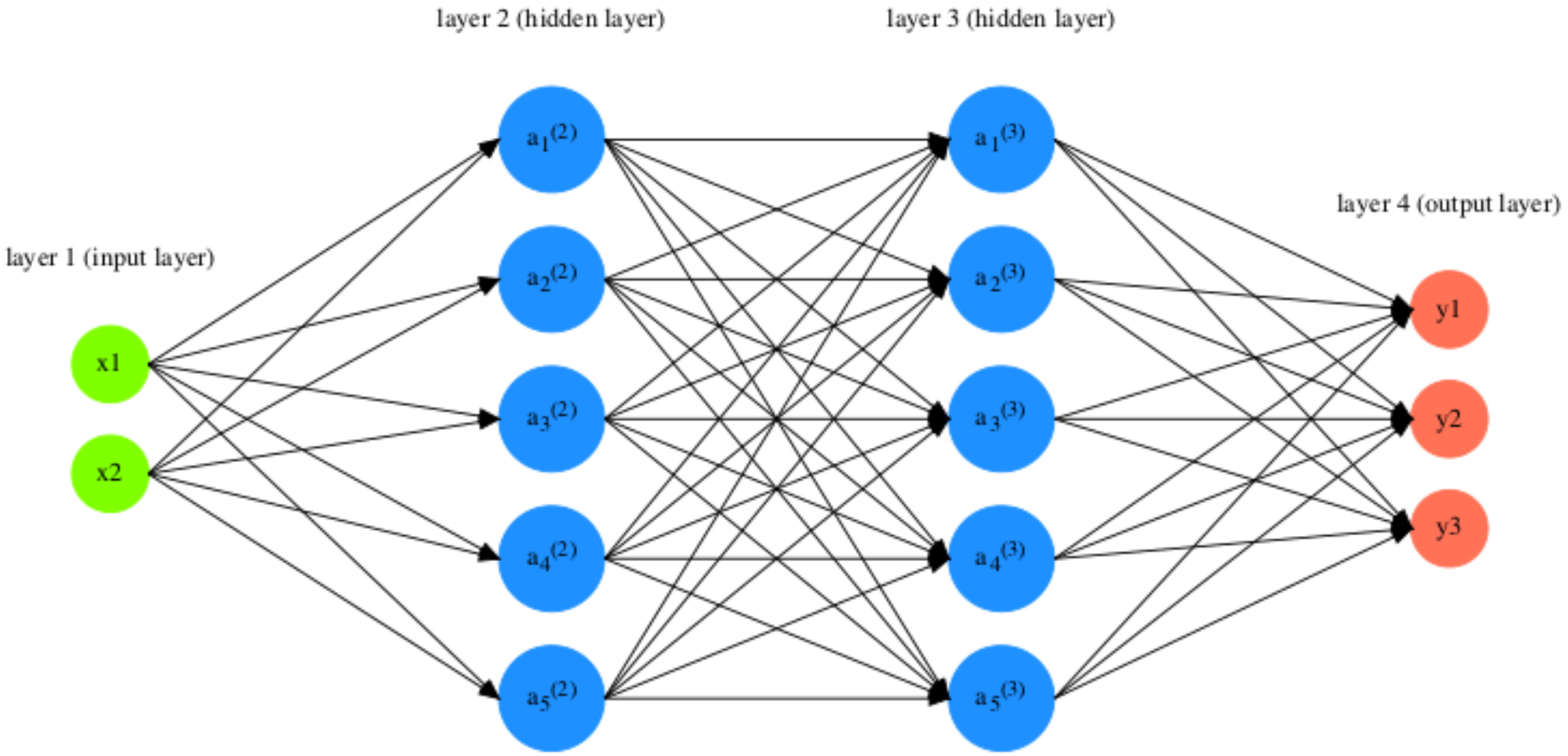




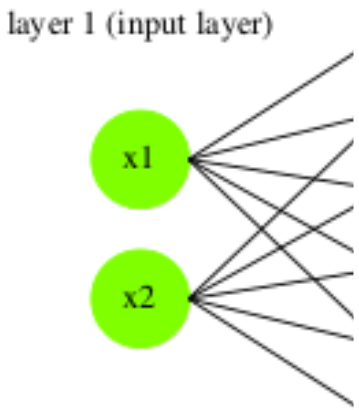


A rectified linear unit “ReLU”
Figure: Goodfellow, Bengio and Courville

Neural Networks



Forward Pass



Forward Pass

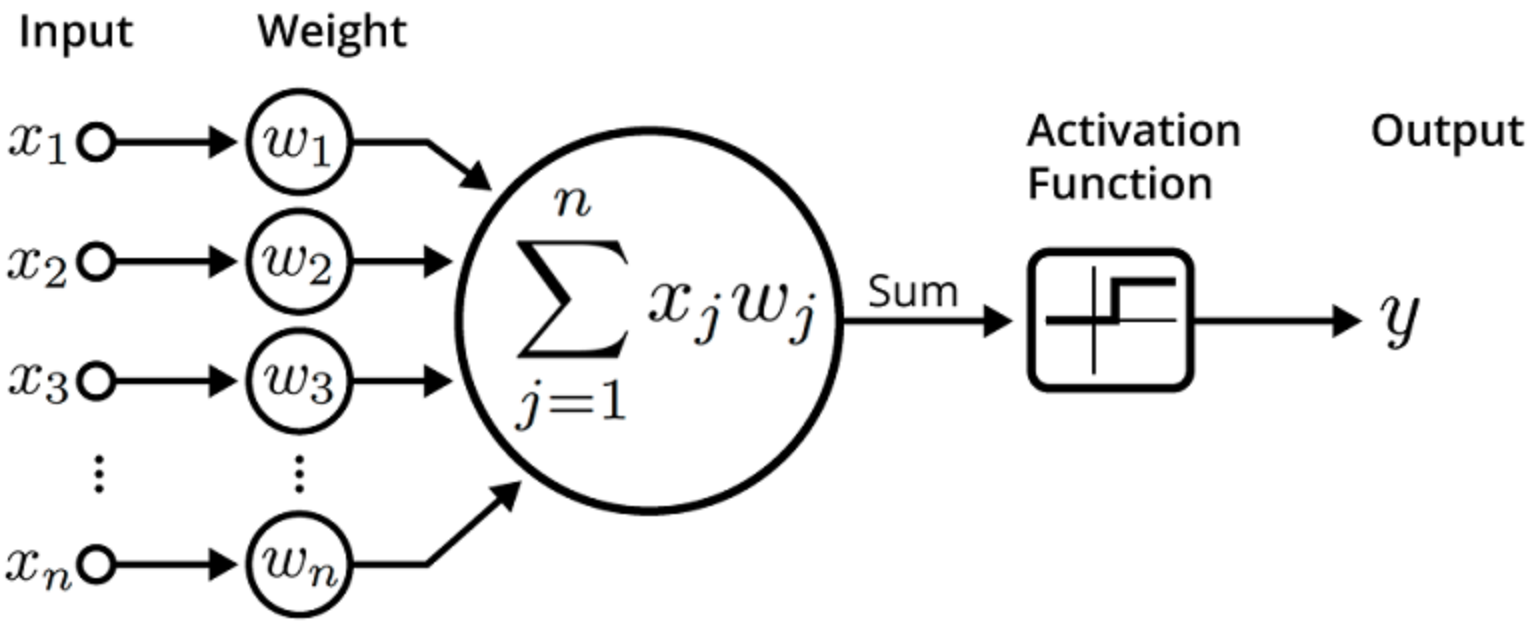
Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Figure: Goodfellow, Bengio and Courville

Training a Neuron



$$\underset{w_1,\ldots,w_n}{\operatorname{argmin}} \sum_{(x,y)\in Data} L(y, f(x))$$

Gradient Descent

- $\text{argmin}_{w_1,\ldots,w_n} \sum_{(x,y)\in Data} L(y, f(x))$

- $\nabla_{\mathrm{w}} \sum_{(x,y)\in Data} L(y, f(x)) = \sum_{(x,y)\in Data} \nabla_{\mathrm{w}} L(y, f(x))$
- $= \sum_{(x,y)\in Data} \partial_{f(x)}[L(y, f(x))]\nabla_w f(x)$

- $w = w - \alpha \sum_{(x,y)\in Data} \partial_{f(x)}[L(y, f(x))]\nabla_w f(x)$

- Stochastic: $w = w - \alpha \partial_{f(x)}[L(y, f(x))]\nabla_w f(x)$
  - In practice use a “minibatch” and more complex updates (e.g. use ADAM)

Chain Rule

- $\nabla_w L\big(y, f(x)\big) = \partial_{f(x)}[L(y, f(x))]\nabla_w f(x)$

- Suppose we have a single input at each layer

- $\dfrac{\partial z}{\partial y}$
- $\dfrac{\partial z}{\partial x} = \dfrac{\partial z}{\partial y}\dfrac{\partial y}{\partial x}$
- $\dfrac{\partial z}{\partial w} = \dfrac{\partial z}{\partial y}\dfrac{\partial y}{\partial w} = \dfrac{\partial z}{\partial y}\dfrac{\partial y}{\partial x}\dfrac{\partial x}{\partial w}$

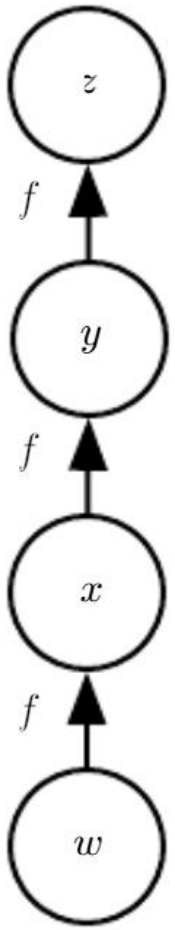


Figure: Goodfellow, Bengio and Courville

Backward Pass

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses, in addition to the input $\boldsymbol{x}$, a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$

end for

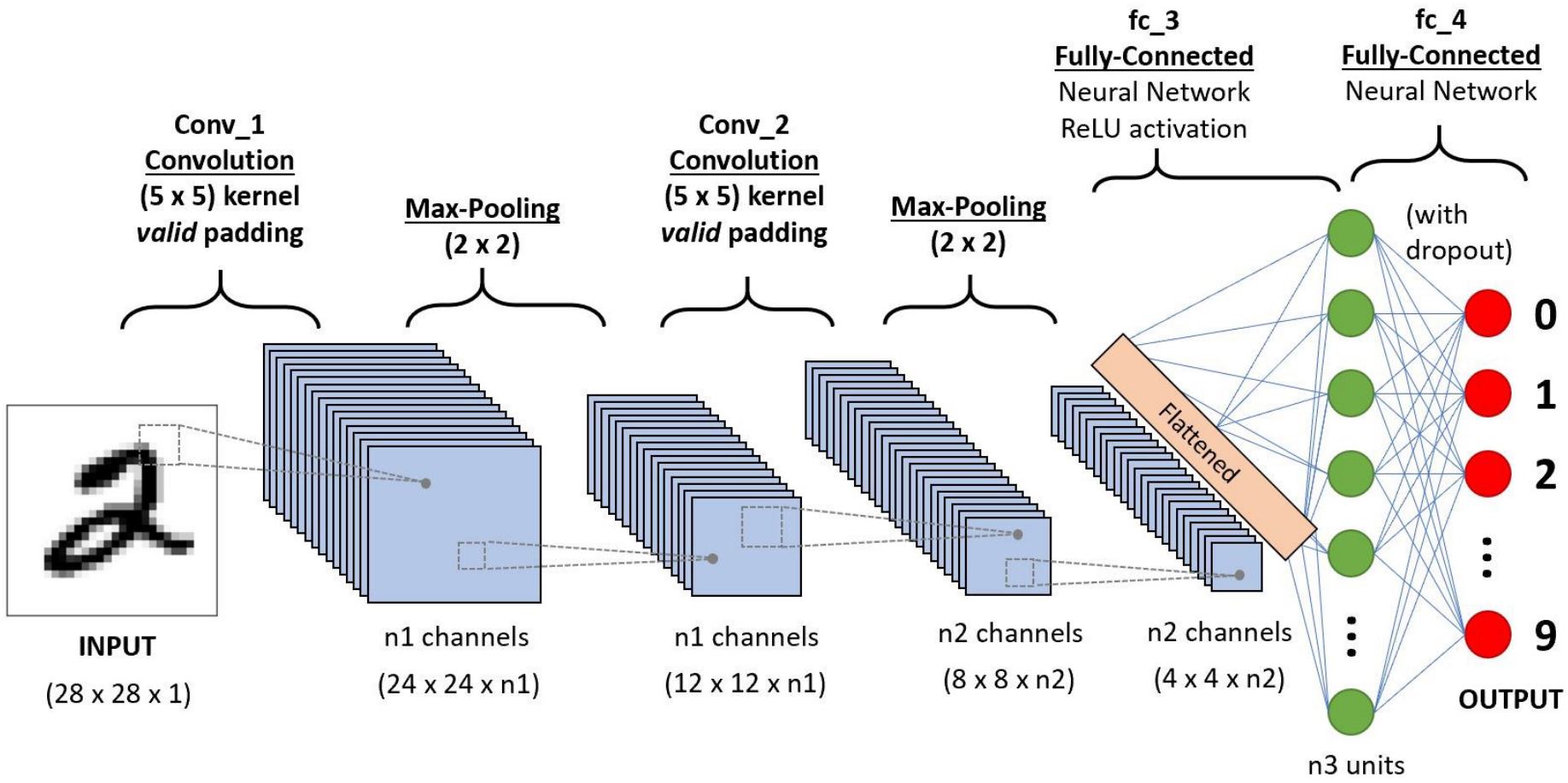
Figure: Goodfellow, Bengio and Courville

Vanishing and Exploding Gradients

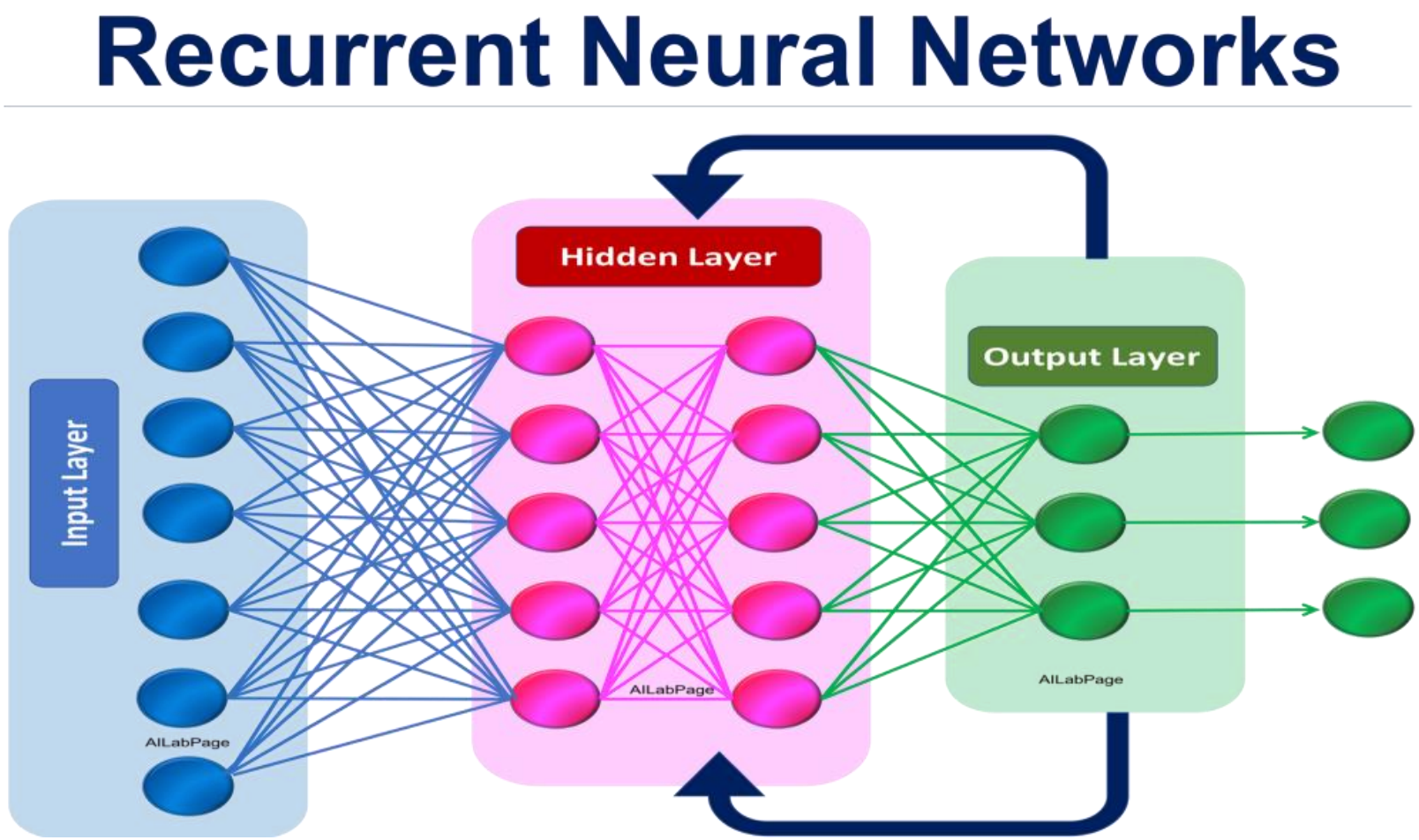
- Consider the gradient for a really deep network
- $\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}\frac{\partial w}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial t}\frac{\partial t}{\partial s}\frac{\partial s}{\partial r}\frac{\partial r}{\partial q}\frac{\partial q}{\partial p}\frac{\partial p}{\partial o}\frac{\partial o}{\partial n}\frac{\partial n}{\partial m}\frac{\partial m}{\partial l}\frac{\partial l}{\partial k}\frac{\partial k}{\partial j}\frac{\partial j}{\partial i}\frac{\partial i}{\partial h}\ldots$
- What might go wrong?
- Possible Solutions:
  - Add “skip” connections deeper into network (ResNets)
  - ReLU
  - Normalization / Regularization
  - Clipping

Neural Networks for RL

Convolutional Neural Networks



Recurrent Neural Networks



A Bad Example

RL Challenge 1: Deadly Triad

- Function Approximation
- TD-Methods
- Off Policy Training

RL Challenge 2: Semi-Gradient

- $VE(W) = \sum_s \mu(s)[r_{t+1} + \gamma\, \hat{v}(s', w) - \hat{v}(s, w)]^2$
- $\nabla VE(W) = \sum_s \mu(s) 2[r_{t+1} + \gamma\, \hat{v}(s', w) - \hat{v}(s, w)][-\nabla \hat{v}(s, w)]$

RL Challenge 3: Correlated Data

- $s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, \ldots$

RL Challenge 4: Moving Target

- $VE(W) = \sum_s \mu(s)[r_{t+1} + \gamma\, \hat{v}(s', w) - \hat{v}(s, w)]^2$

- $\delta_t = r + \gamma \hat{v}(s', w) - \hat{v}(s, w)$

Summary

- Deep Neural Networks
  - Approximate complex functions
  - Updates are just chain rule
  - Choose structure based on domain features
- Challenges
  - Standard training issues for DNNs
  - RL-specific problems
- Monday: DQN