From Shallow to Deep Language Representations

1 Basics · 2 Shallow Models · 3 Transformers · 4 BERT

KDD'19 Anchorage

Aston Zhang, Haibin Lin, Leonard Lausen, Sheng Zha, Alex Smola

www.d2l.ai gluon-nlp.mxnet.io

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Outline (Basics)

- Installation
- Arrays
- Autograd
- Multilayer Perceptron
- Convolutional Neural Networks
- Basic Sequence Models

Installation

Step 1 - Conda and Supporting Packages

- Get **miniconda** from repo.continuum.io/packages
- Install Conda packages and create environment

```
conda create --name d2l
conda activate d2l
conda install pip jupyter rise
```

Step 2 - MXNet and Supporting Packages

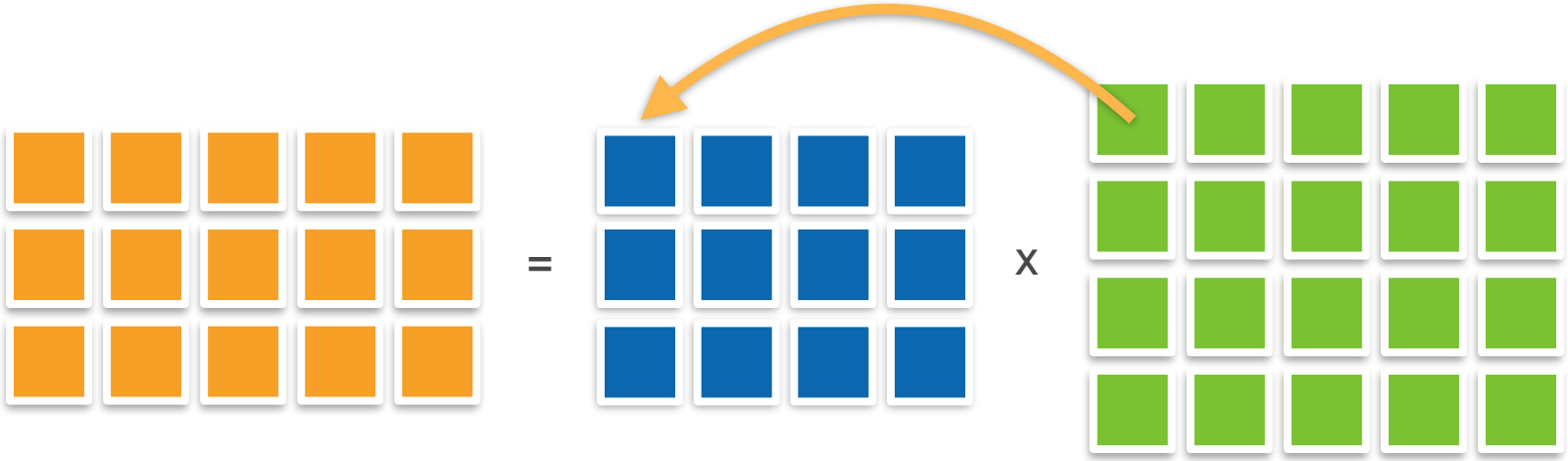
- Install nvidia.com/cuda version 10.1
- Install MXNet packages (mxnet or mxnet-cu101)

```
pip install mxnet==1.5.0
pip install d2l==0.8.2
pip install gluonnlp==0.7.1
pip install nltk sacremoses
```

Step 3 - Gluon-NLP and Tutorial

```
wget https://github.com/astonzhang/KDD19-tutorial/archive/master.zip
unzip master.zip
jupyter notebook
```

Ndarray



numpy.d2l.ai

NDArray (Linear Algebra on GPUs in Python)

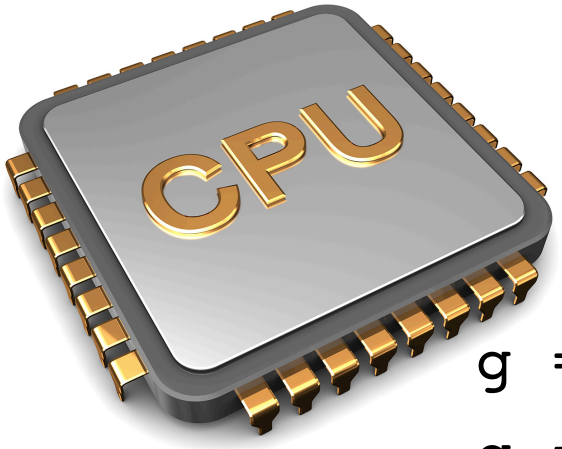
- **NumPy**
  - Limited to CPUs
  - No automatic differentiation
  - **Blocking (returns only once computation is performed)**
- **NDArray**
  - Multiple CPUs / GPUs via device context
  - Distributed systems in the cloud
  - **Nonblocking C++ backend (no Python GIL)**
  - **Lazy evaluation**

amazon web services

NDArray Device Contexts

```
context = mx.cpu()
```

```
context = mx.gpu(0)
context = mx.gpu(1)
```



```
g = copyto(c)
g = c.as_in_context(mx.gpu(0))
```

amazon web services

Automatic Differentiation

Automatic Differentiation

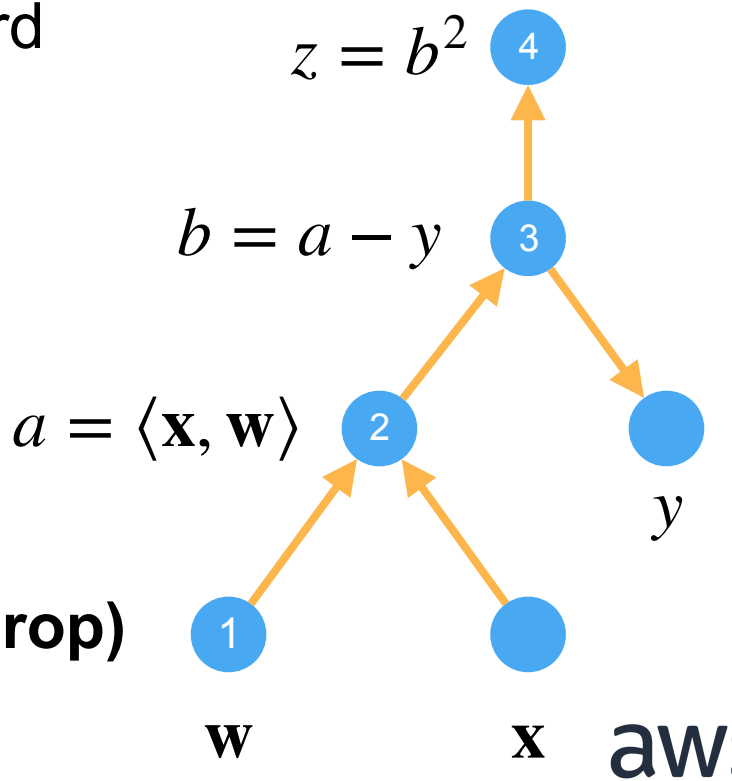
$$z = \left(\langle \mathbf{x}, \mathbf{w} \rangle - y\right)^2$$

Computing derivatives by hand is hard

Computers can automate this

Compute graph

- Build explicitly
  (TensorFlow, MXNet Symbol)
- **Build implicitly by tracing**
  (Chainer, PyTorch, DeepNumpy)



Chain rule (evaluate e.g. via backprop)

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u_n} \frac{\partial u_n}{\partial u_{n-1}} \cdots \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial x}$$

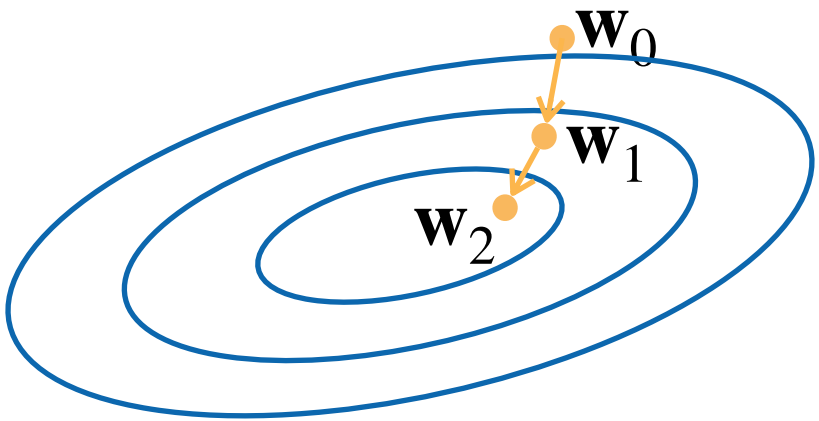
Stochastic
Gradient
Descent
momentum
negative
gradient

Gradient Descent

Choose a starting point $\mathbf{w}_0$

Repeat to update weight

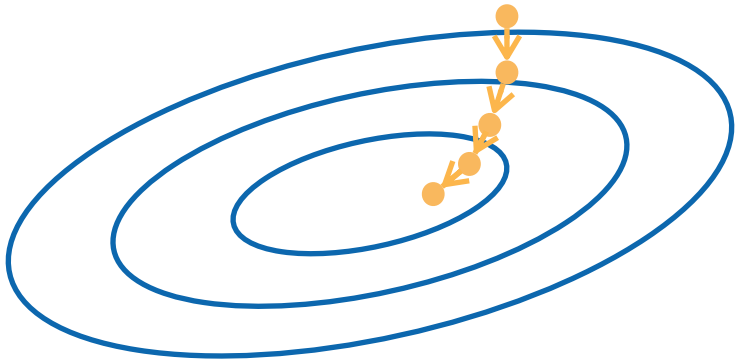
$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \partial_w l(w_{t-1})$$



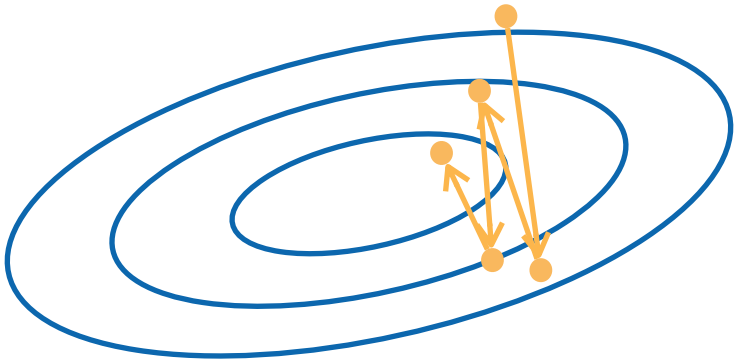
- Gradient direction indicates increase in value
- Learning rate adjusts step length

Goldilocks Learning Rate

Too small



Too big



Stochastic Gradient Descent (SGD)

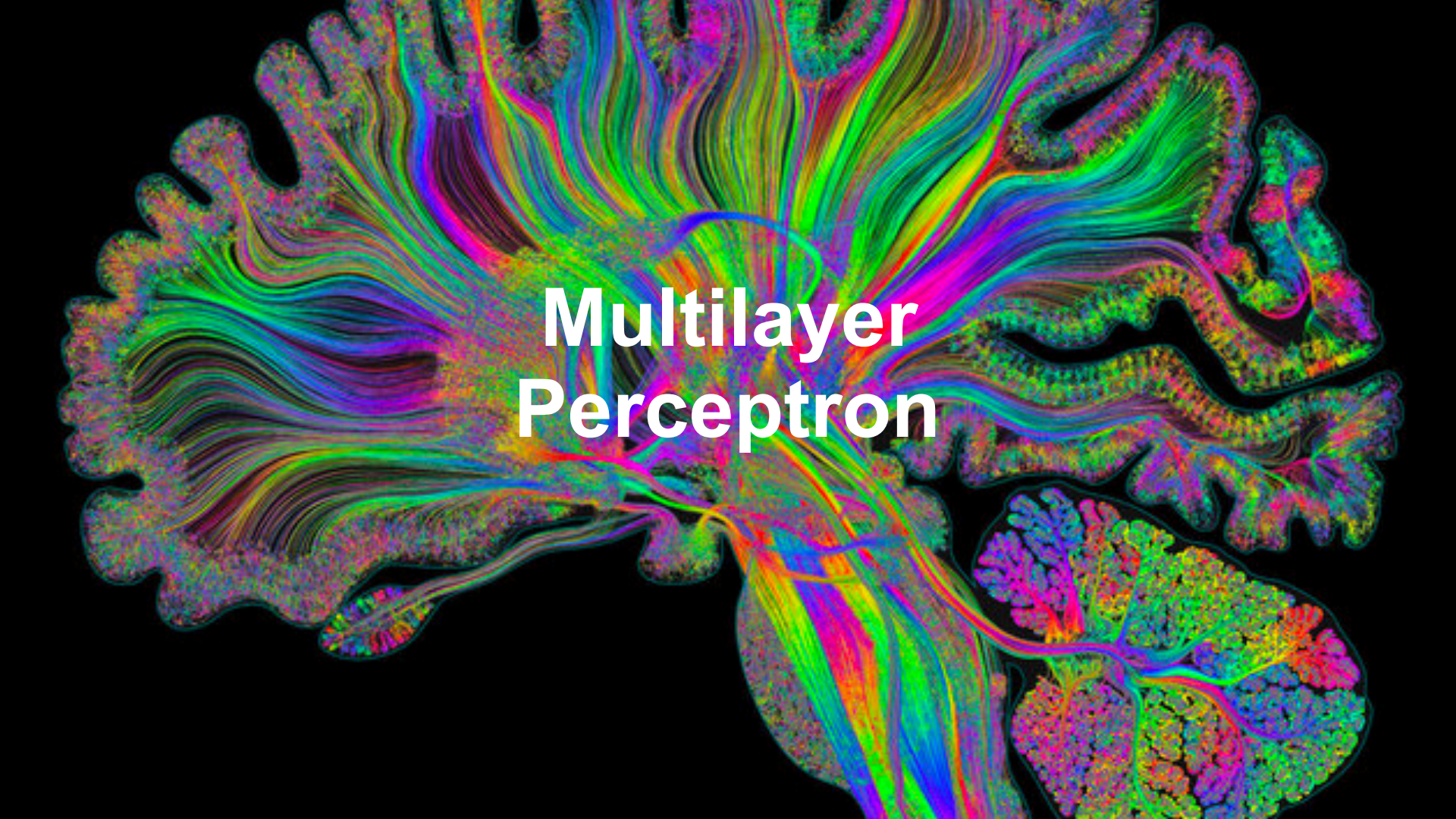
- Computing the gradient over all data is too slow
- Redundancy in data (e.g. many similar digits)

- Single observation is not efficient on GPU

- Sample b examples $i_1, \ldots i_b$ to approximate loss/gradient

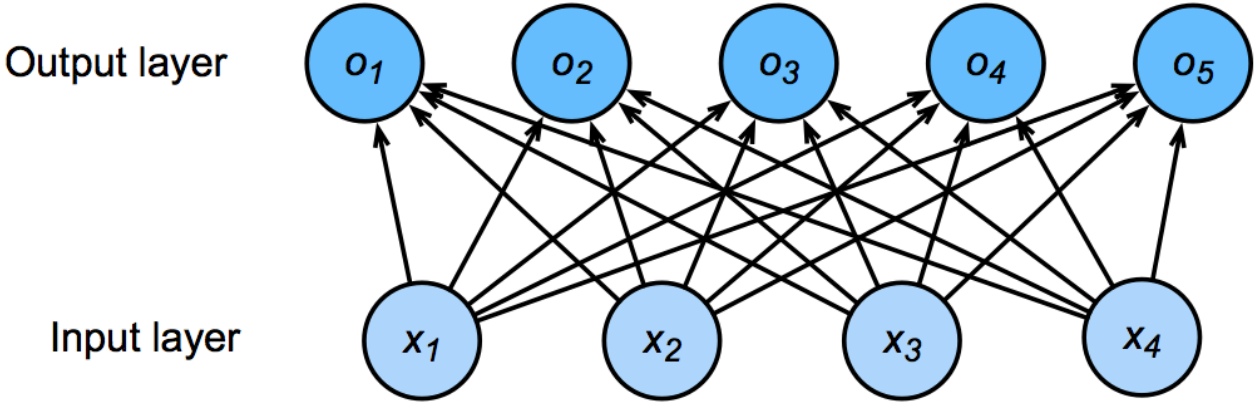
$$\frac{1}{b}\sum_{i \in I_b} l(\mathbf{x}_i, y_i, \mathbf{w}) \text{ and } \frac{1}{b}\sum_{i \in I_b} \partial_{\mathbf{w}} l(\mathbf{x}_i, y_i, \mathbf{w})$$

b is the mini-batch size (chosen for GPU efficiency)

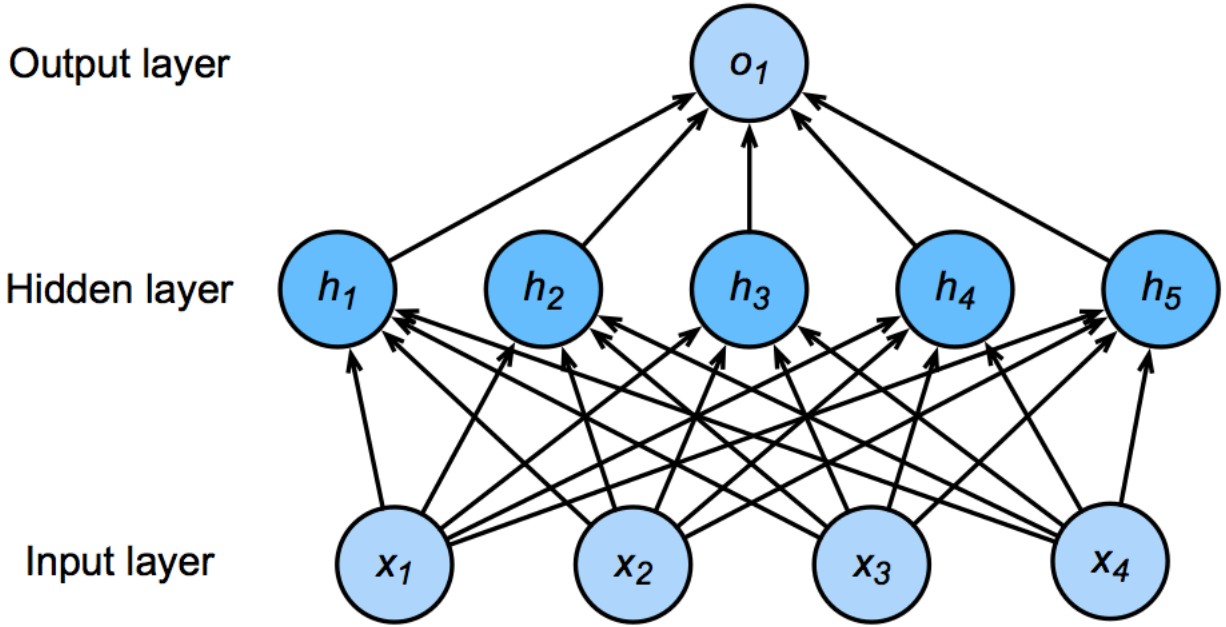


Multilayer Perceptron

Linear Model



Single Hidden Layer



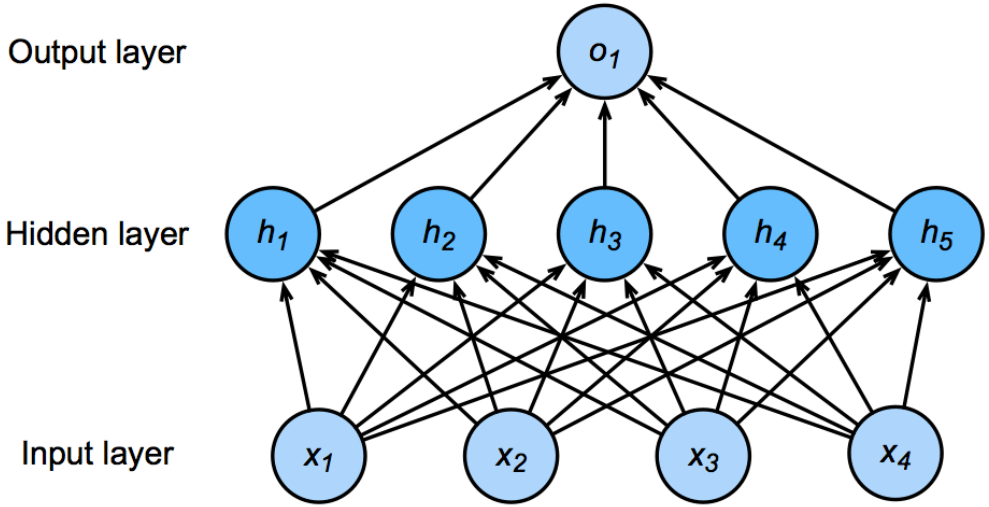
Single Hidden Layer

- Input $\mathbf{x} \in \mathbb{R}^n$
- Hidden $\mathbf{W}_1 \in \mathbb{R}^{m \times n}, \mathbf{b}_1 \in \mathbb{R}^m$
- Output $\mathbf{w}_2 \in \mathbb{R}^m, b_2 \in \mathbb{R}$

$$\mathbf{h} = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{o} = \mathbf{w}_2^T \mathbf{h} + b_2$$

σ is an element-wise activation function



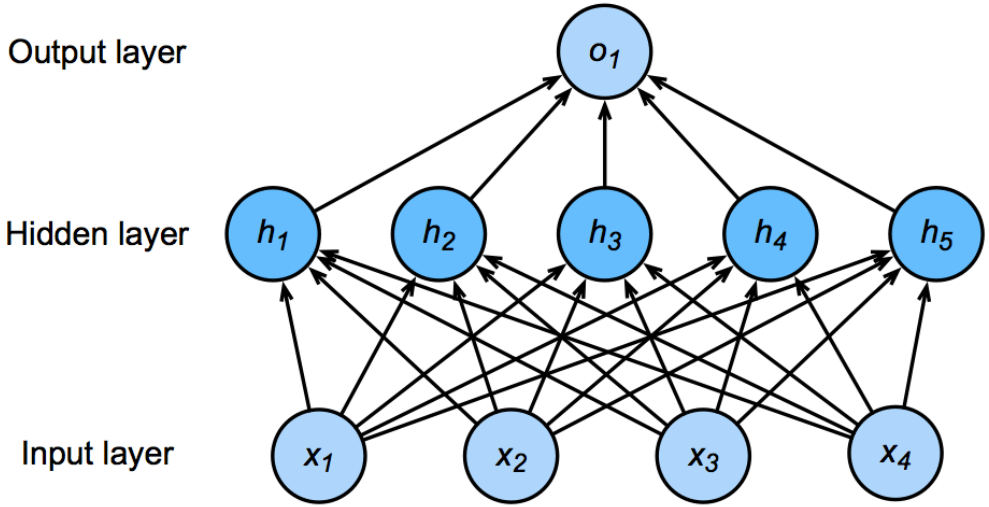
Single Hidden Layer

Why do we need an a nonlinear activation?

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{o} = \mathbf{w}_2^T\mathbf{h} + b_2$$

σ is an element-wise activation function



Single Hidden Layer

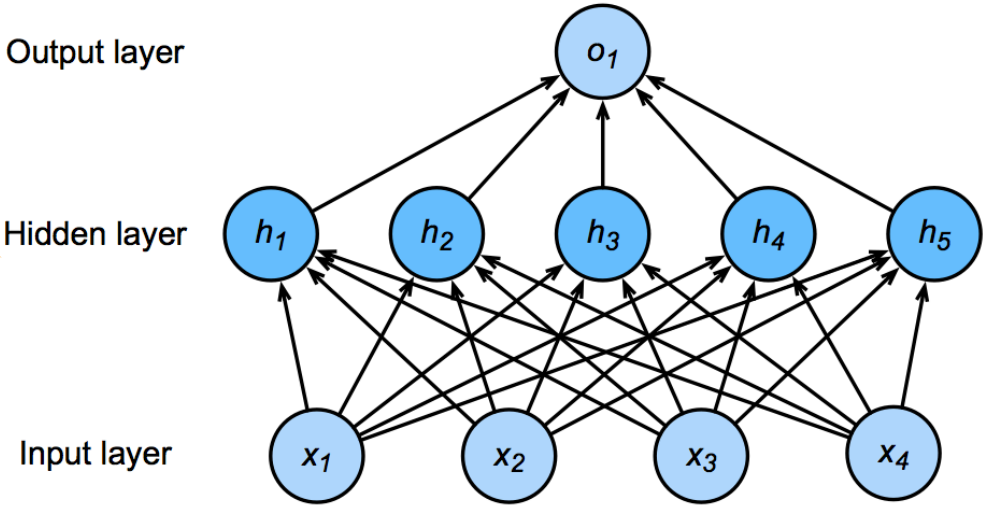
Why do we need an a nonlinear activation?

$$\mathbf{h} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$

$$\mathbf{o} = \mathbf{w}_2^T\mathbf{h} + b_2$$

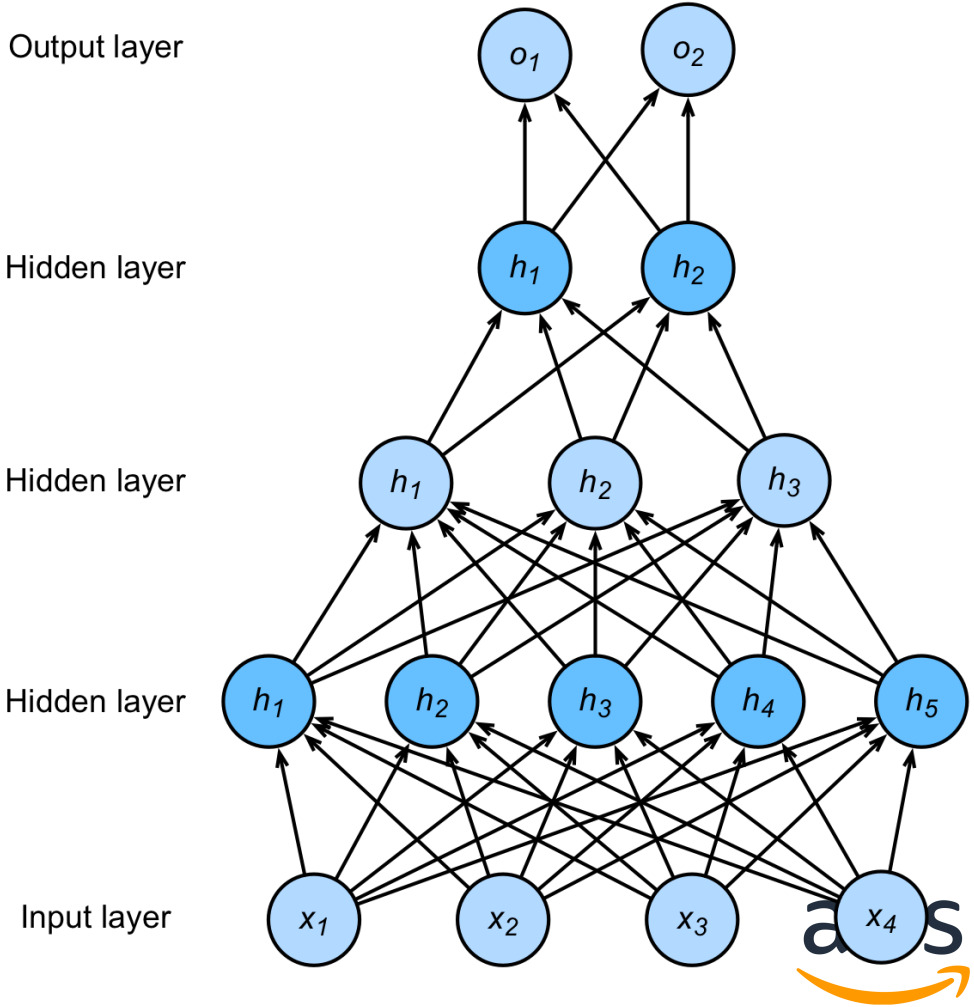
hence $o = \mathbf{w}_2^\top\mathbf{W}_1\mathbf{x} + b'$

Linear ...



Multiple Hidden Layers

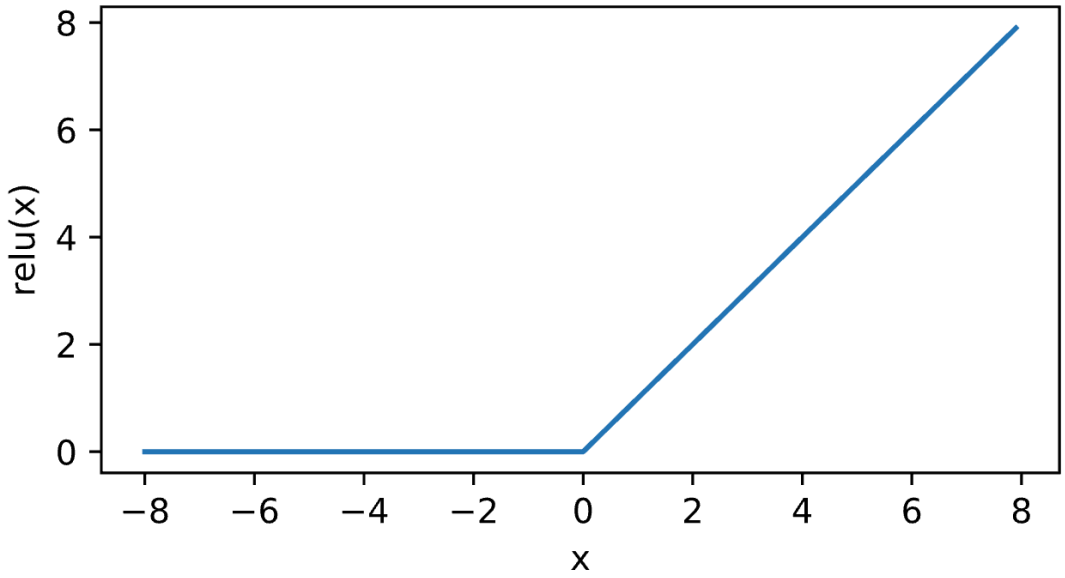
$$\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$
$$\mathbf{h}_3 = \sigma(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$
$$\mathbf{o} = \mathbf{W}_4\mathbf{h}_3 + \mathbf{b}_4$$



ReLU Activation

ReLU: rectified linear unit

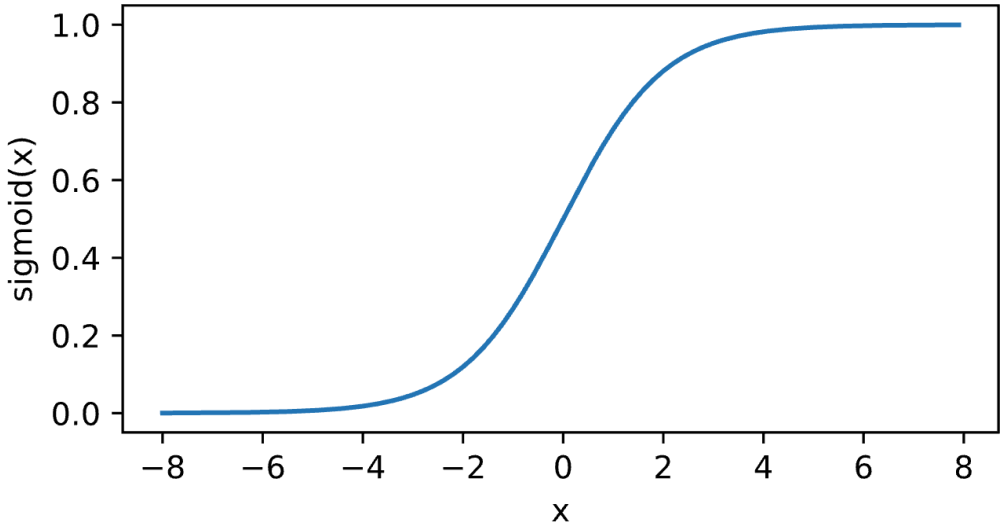
$$\mathrm{ReLU}(x) = \max(x, 0)$$



Sigmoid Activation

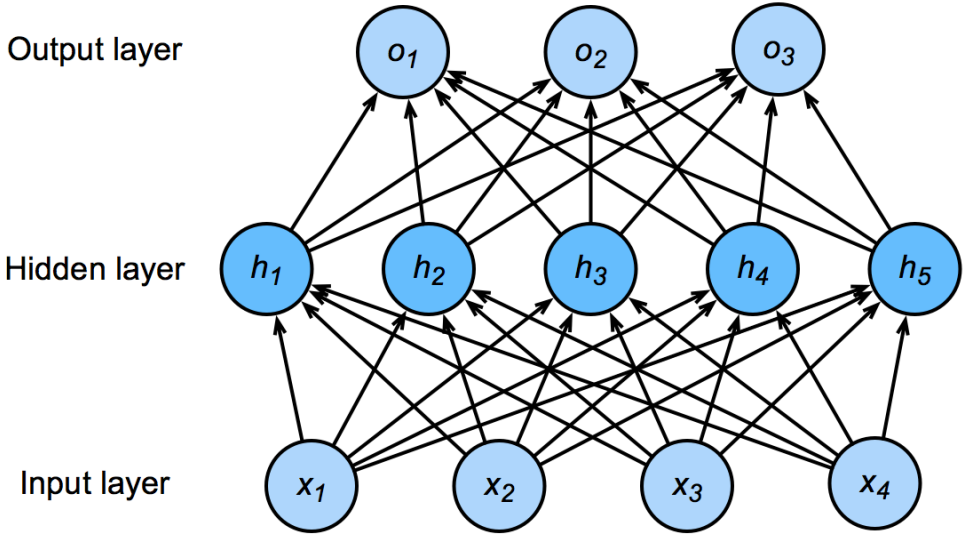
Map input into (0, 1), a soft version of $\sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

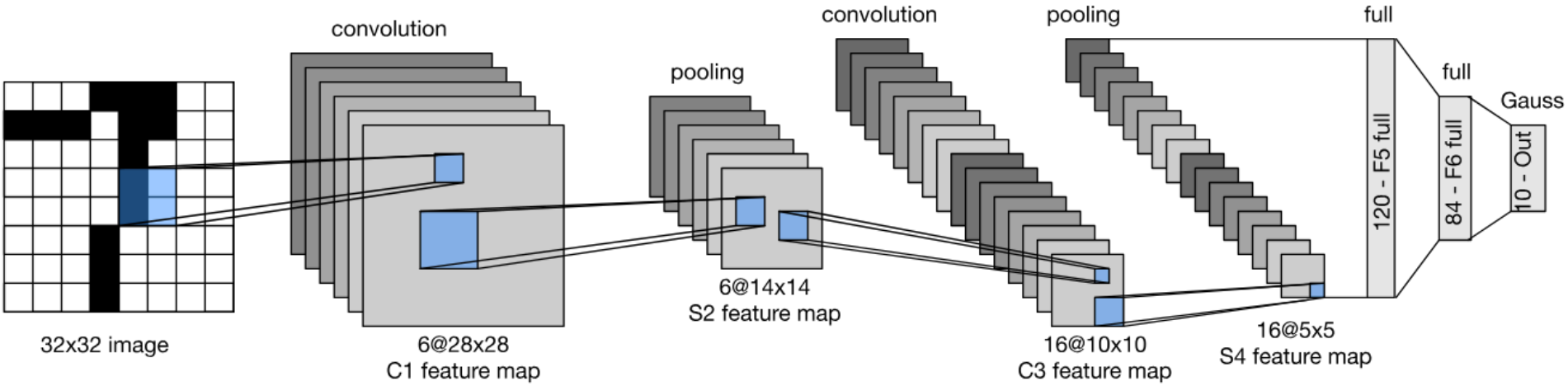


Multiclass Classification

$$y_1, y_2, \ldots, y_k = \text{softmax}(o_1, o_2, \ldots, o_k)$$

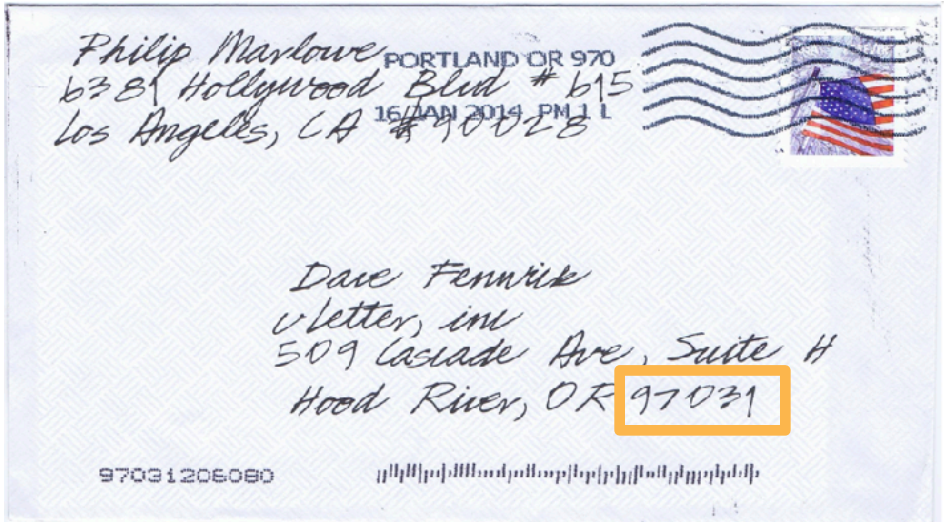


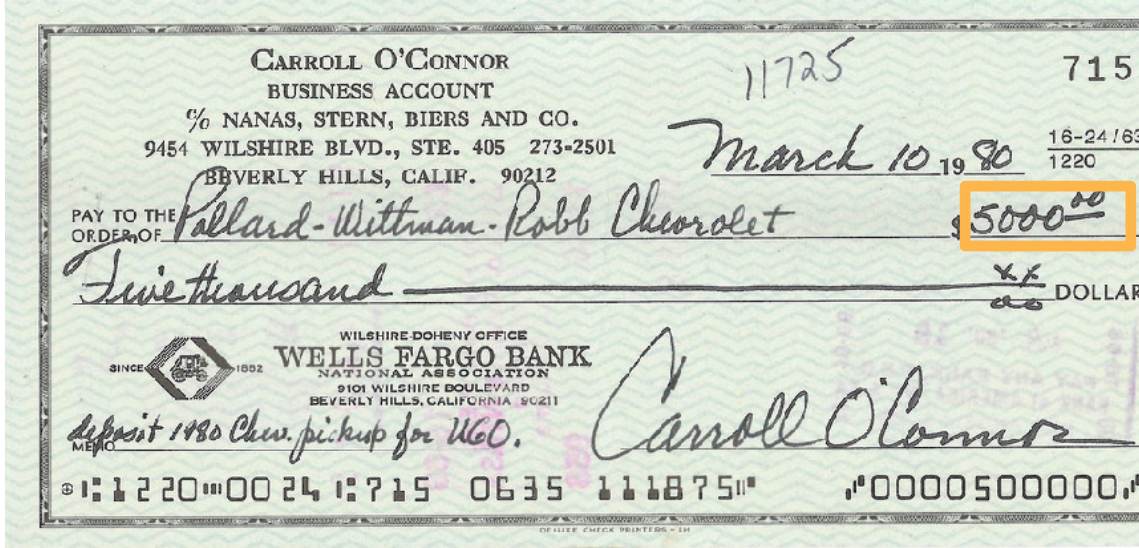
Convolutional Networks - LeNet



numpy.d2l.ai

Handwritten Digit Recognition



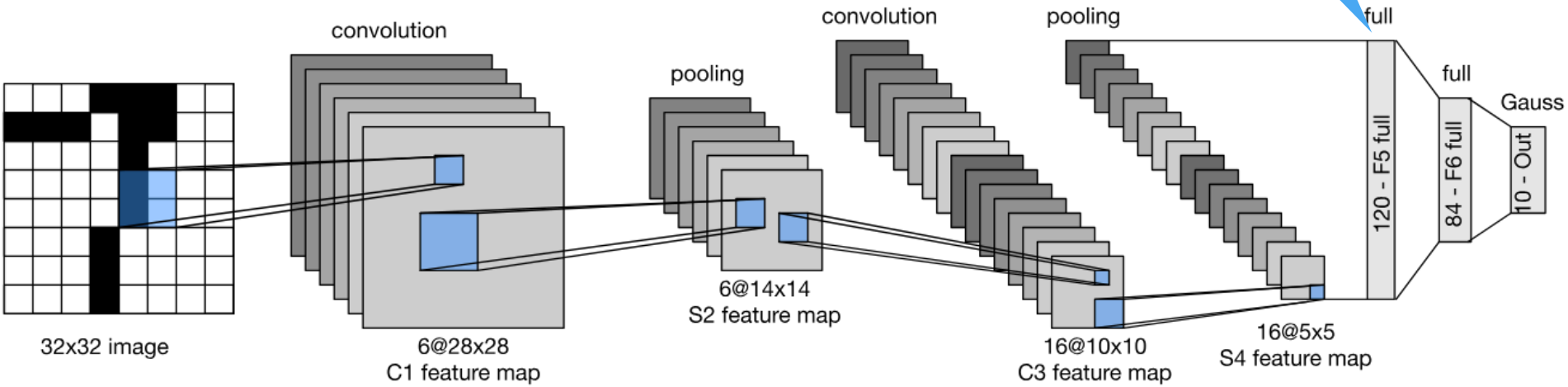


MNIST

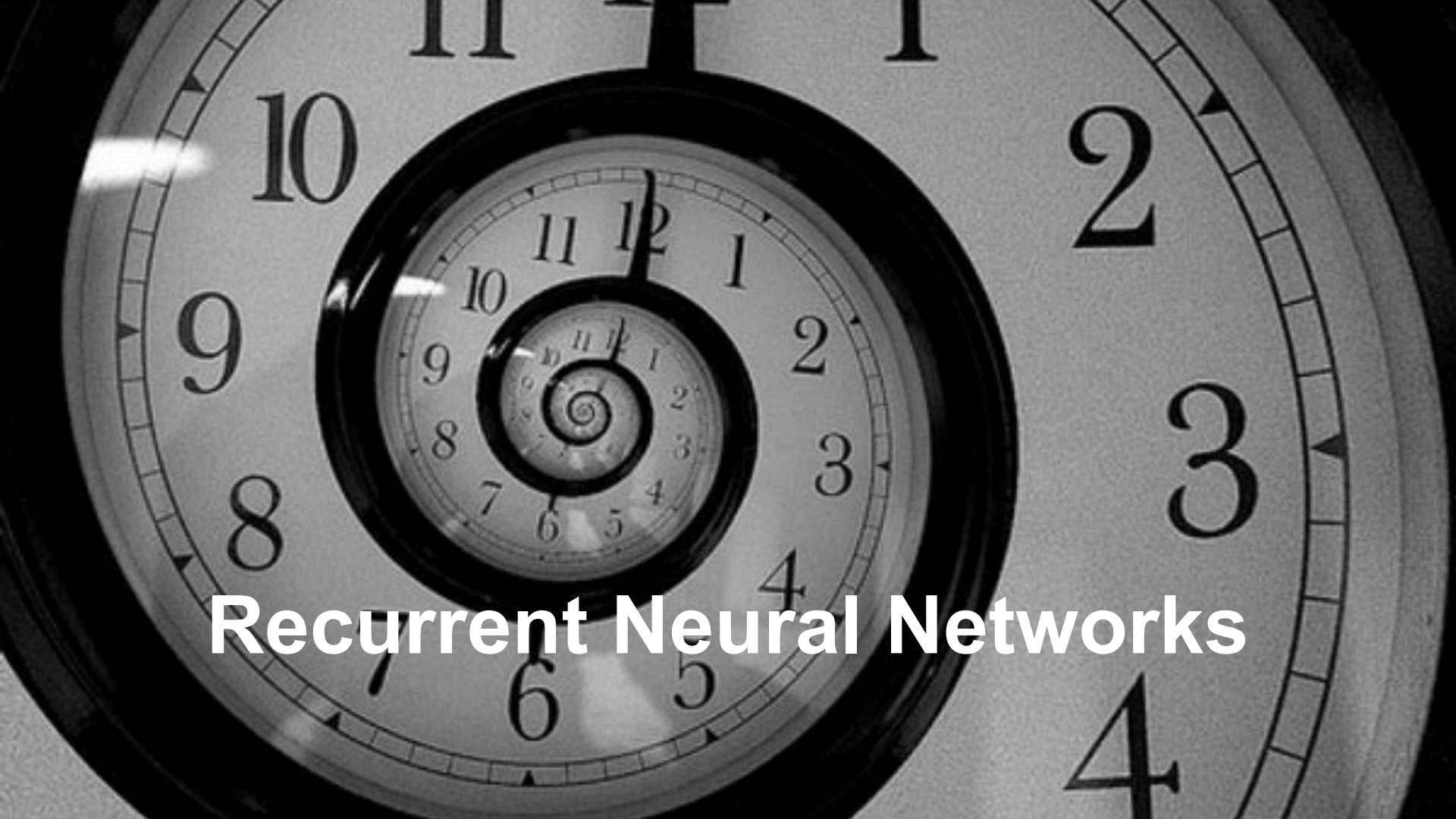
- Centered and scaled
- 60,000 training data
- 10,000 test data
- 28 x 28 images
- 10 classes

Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, 1998 Gradient-based learning applied to document recognition

Expensive if we have many outputs

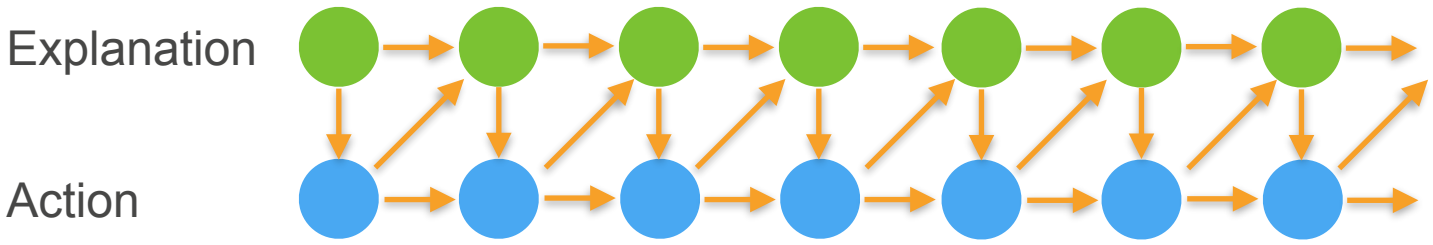


mxnet.io

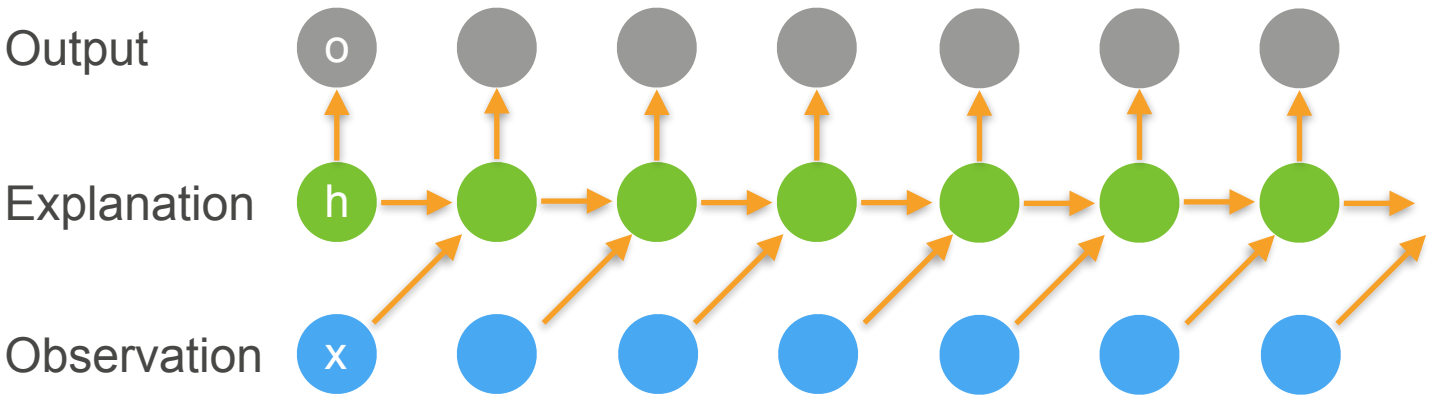


Recurrent Neural Networks

Recurrent Neural Networks (with hidden state)



Recurrent Neural Networks (with hidden state)



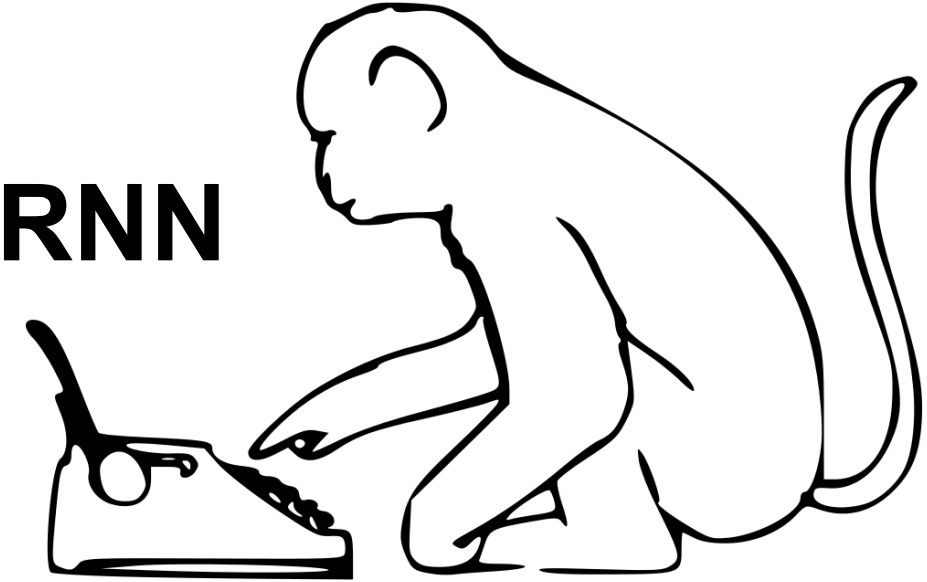
- Hidden State update

$$\mathbf{h}_t = \phi(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{hx}\mathbf{x}_{t-1} + \mathbf{b}_h)$$

- Observation update

$$\mathbf{o}_t = \phi(\mathbf{W}_{ho}\mathbf{h}_t + \mathbf{b}_o)$$

Implementing an RNN Language Model



Input Encoding

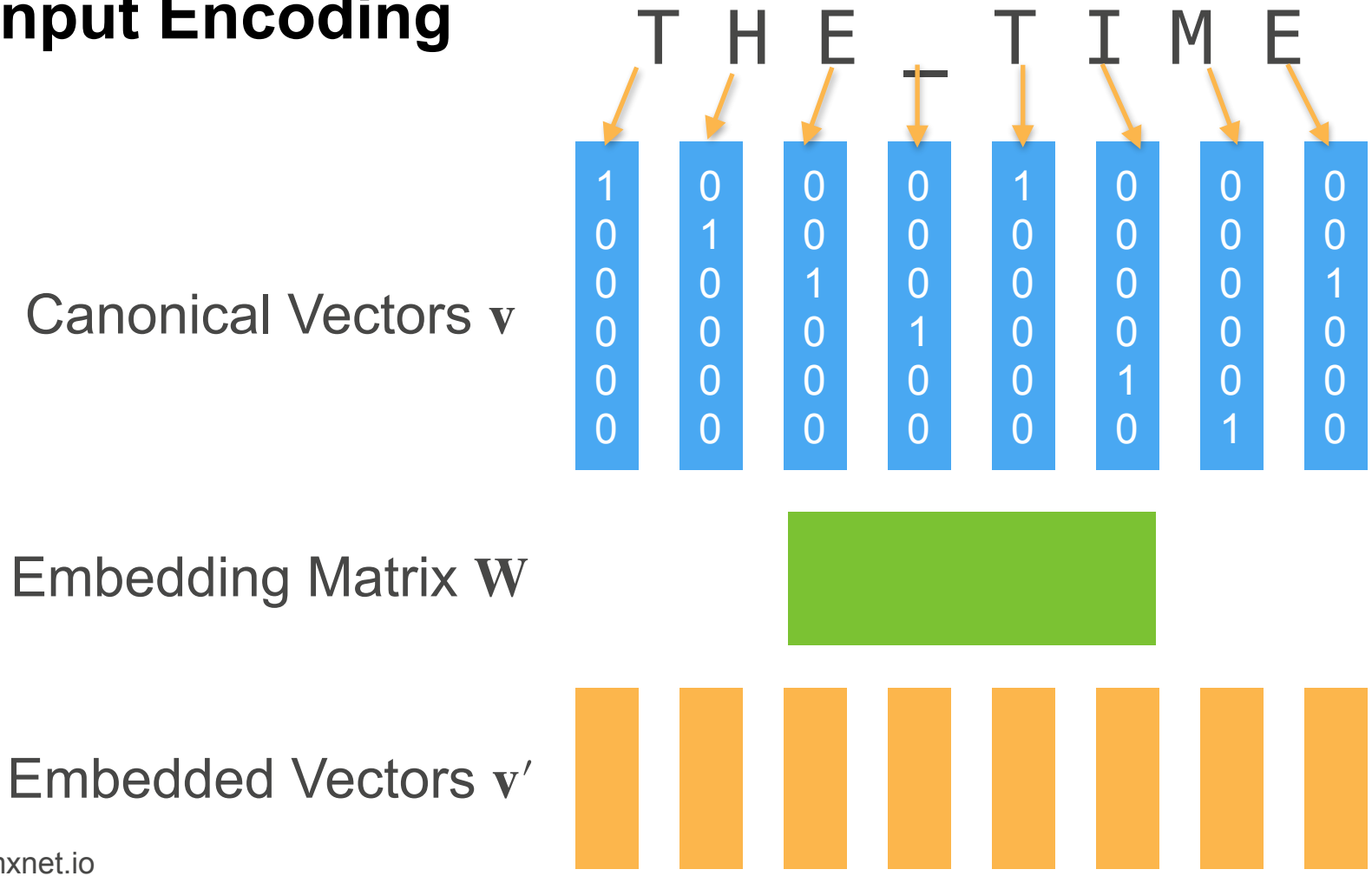
- Text - need to map input tokens to vectors
  - Pick granularity (words, characters, subwords)
  - Map to indicator vectors

```
nd.one_hot(nd.array([0, 2]), vocab_size)
```

```
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
<NDArray 2x43 @cpu(0)>
```

- Multiply by embedding matrix W

Input Encoding



RNN with hidden state mechanics

- Input
  vector sequence $\mathbf{x}_1, \ldots, \mathbf{x}_T$
- Hidden States
  vector sequence $\mathbf{h}_1, \ldots, \mathbf{h}_T$ where $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$
- Output
  vector sequence $\mathbf{o}_1, \ldots, \mathbf{o}_T$ where $\mathbf{o}_t = g(\mathbf{h}_t)$

Read sequence to generate hidden states, then start generating outputs. Often outputs (symbols) are used as input for next hidden state (and thus output).

Output Decoding

Output Vectors $\mathbf{o}$

Decoding Matrix $\mathbf{W}'$

$$p(y \mid \mathbf{o}) \propto \exp\left(\mathbf{v}_y^\top \mathbf{o}\right) = \exp(\mathbf{o}[y])$$

One-hot decoding

Gradients

- Long chain of dependencies for backprop
  - Need to keep a lot of intermediate values in memory
  - Butterfly effect style dependencies
  - Gradients can vanish or diverge (more on this later)
- Clipping to prevent divergence

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}$$

rescales to gradient of size at most θ

Perplexity

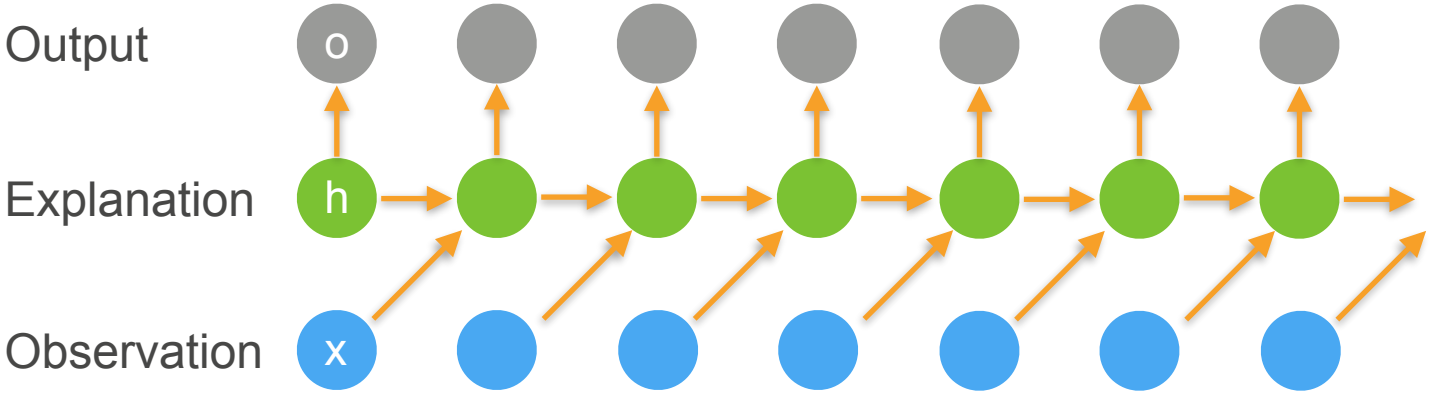
- Typically measure accuracy with log-likelihood
  - This makes outputs of different length incomparable (e.g. bad model on short output has higher likelihood than excellent model on very long output)
  - Normalize log-likelihood to sequence length

$$-\sum_{t=1}^{T} \log p(y_t \,|\, \text{model}) \quad \text{vs.} \quad \pi := -\frac{1}{T}\sum_{t=1}^{T} \log p(y_t \,|\, \text{model})$$

- Perplexity is exponentiated version $\exp(\pi)$ (effectively number of possible choices on average)

Truncated Backprop Through Time

Recurrent Neural Networks (with hidden state)



- Hidden State update

$$h_t = f(h_{t-1}, x_{t-1}, w)$$

- Observation update

$$o_t = g(h_t, w)$$

Objective function

- RNN generates output which needs to be compared to target labels

$$L(x, y, w) = \sum_{t=1}^{T} l(y_t, o_t)$$

- Gradient

$$\partial_w L = \sum_{t=1}^{T} \partial_w l(y_t, o_t)$$

$$= \sum_{t=1}^{T} \partial_{o_t} l(y_t, o_t) \left[\partial_w g(h_t, w) + \partial_{h_t} g(h_t, w) \partial_w h_t \right]$$

Latent State Gradient $\partial_w h_t$

- Objective Function

$$\partial_w L = \sum_{t=1}^{T} \partial_w l(y_t, o_t) = \sum_{t=1}^{T} \partial_{o_t} l(y_t, o_t) \left[\partial_w g(h_t, w) + \partial_{h_t} g(h_t, w) \partial_w h_t \right]$$

- Gradient Recursion

$$\partial_w h_t = \partial_w f(x_t, h_{t-1}, w) + \partial_h f(x_t, h_{t-1}, w) \partial_w h_{t-1}$$

$$= \sum_{i=t}^{1} \left[\prod_{j=t}^{i} \partial_h f(x_j, h_{j-1}, w) \right] \partial_w f(x_i, h_{i-1}, w)$$

Latent State Gradient $\partial_w h_t$

- Gradient Recursion

$$\partial_w h_t = \sum_{i=t}^{1} \left[\prod_{j=t}^{i} \partial_h f(x_j, h_{j-1}, w) \right] \partial_w f(x_i, h_{i-1}, w)$$

Too Many Terms

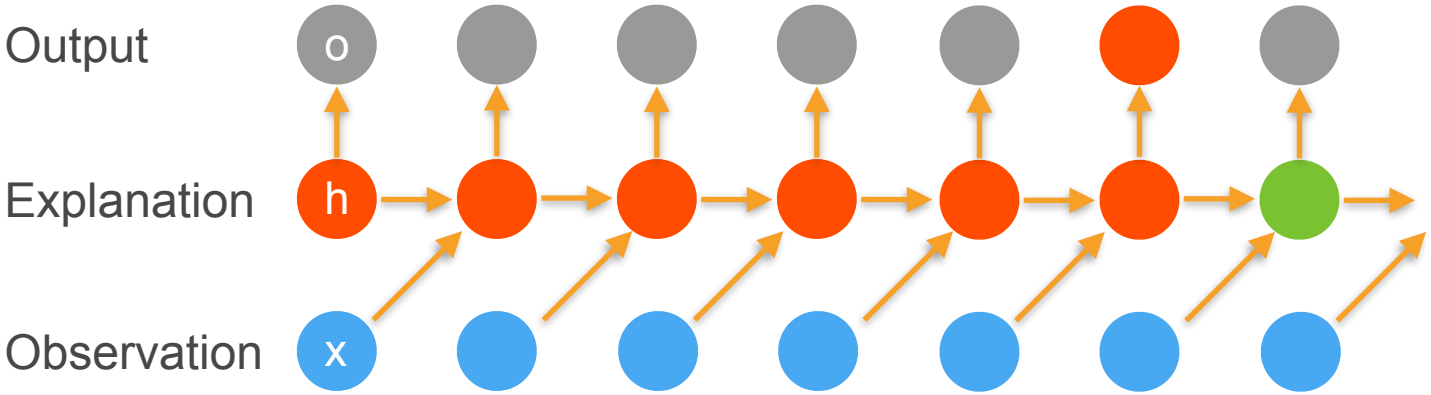
Unstable (divergence)

expensive

Latent State Gradient $\partial_w h_t$

- Gradient Recursion

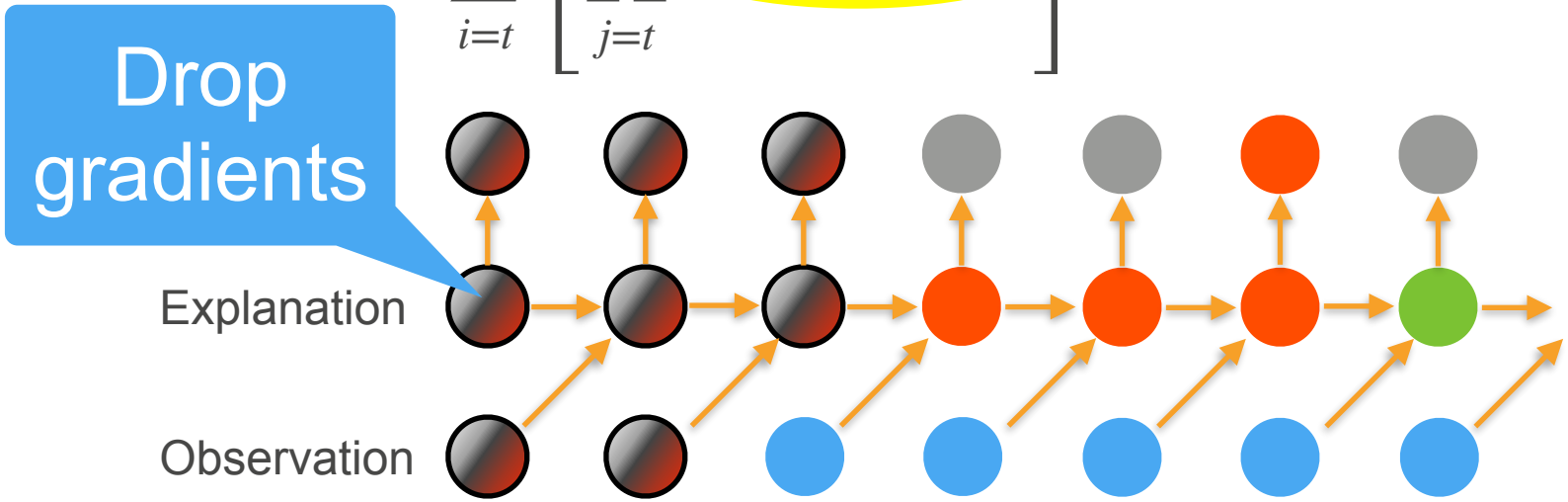
$$\partial_w h_t = \sum_{i=t}^{1} \left[\prod_{j=t}^{i} \partial_h f(x_j, h_{j-1}, w) \right] \partial_w f(x_i, h_{i-1}, w)$$



Latent State Gradient $\partial_w h_t$

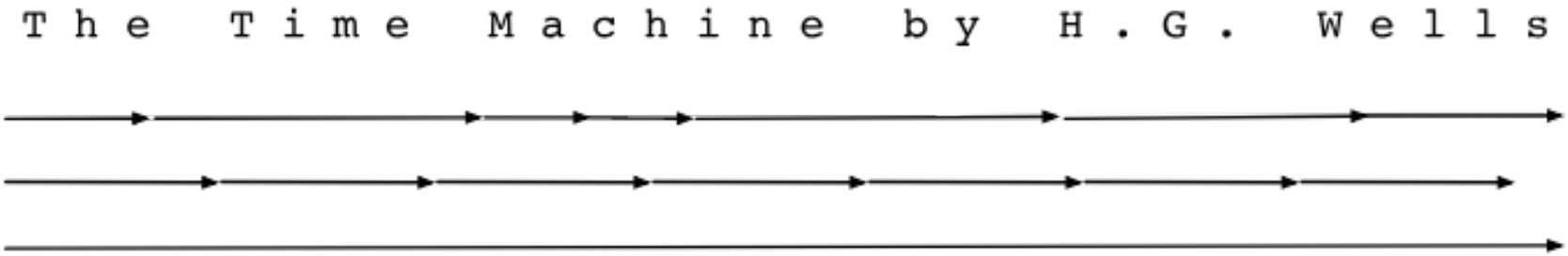
- Gradient Recursion

$$\partial_w h_t = \sum_{i=t}^{1} \left[\prod_{j=t}^{i} \partial_h f(x_j, h_{j-1}, w) \right] \partial_w f(x_i, h_{i-1}, w)$$

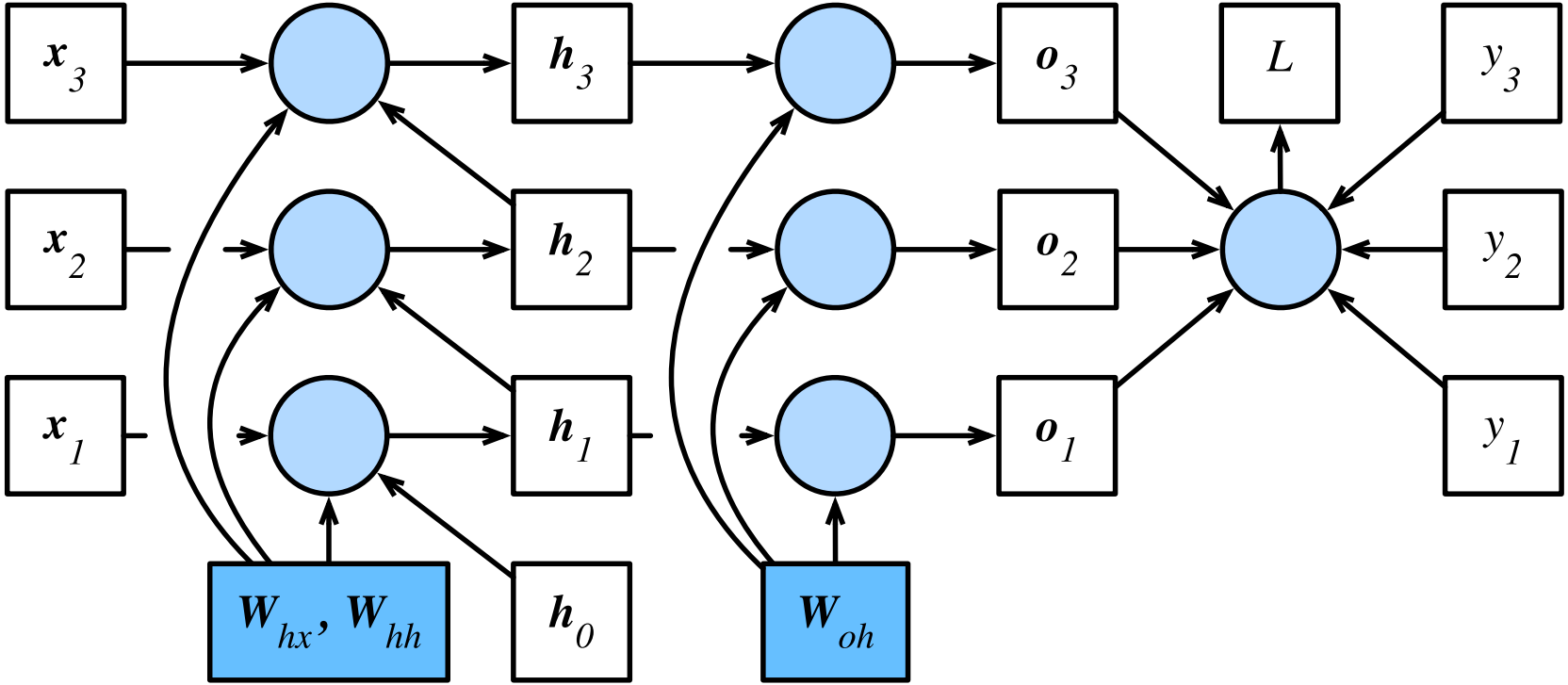


Truncated BPTT

- Don’t truncate (naive strategy, costly and divergent)
- Truncate at fixed intervals
  (standard approach, is approximation but works well)
- ~~Variable length (Tallec and Olivier, 2015)~~
  ~~(is exact after reweighting, doesn’t work better in practice)~~



Computational Graph



Truncation in practice

- Compute forward pass **across** truncation boundaries
- Backprop only until truncation boundary (typically mini batch boundary, too)
- In code

```
for s in state:
    s.detach()
```

- Good reason for why sequential sampling is much more accurate than random - state is carried through.

Summary (Basics)

- Installation
- Arrays
- Autograd
- Multilayer Perceptron
- Convolutional Neural Networks
- Basic Sequence Models