DATA SCIENCE

IN (ASTRO) PARTICLE PHYSICS and COSMOLOGY: the BRIDGE to INDUSTRY

Machine learning: Neural Networks

Márcio Ferreira

Centro de Física da Universidade de Coimbra

27 - 30 June (2022)

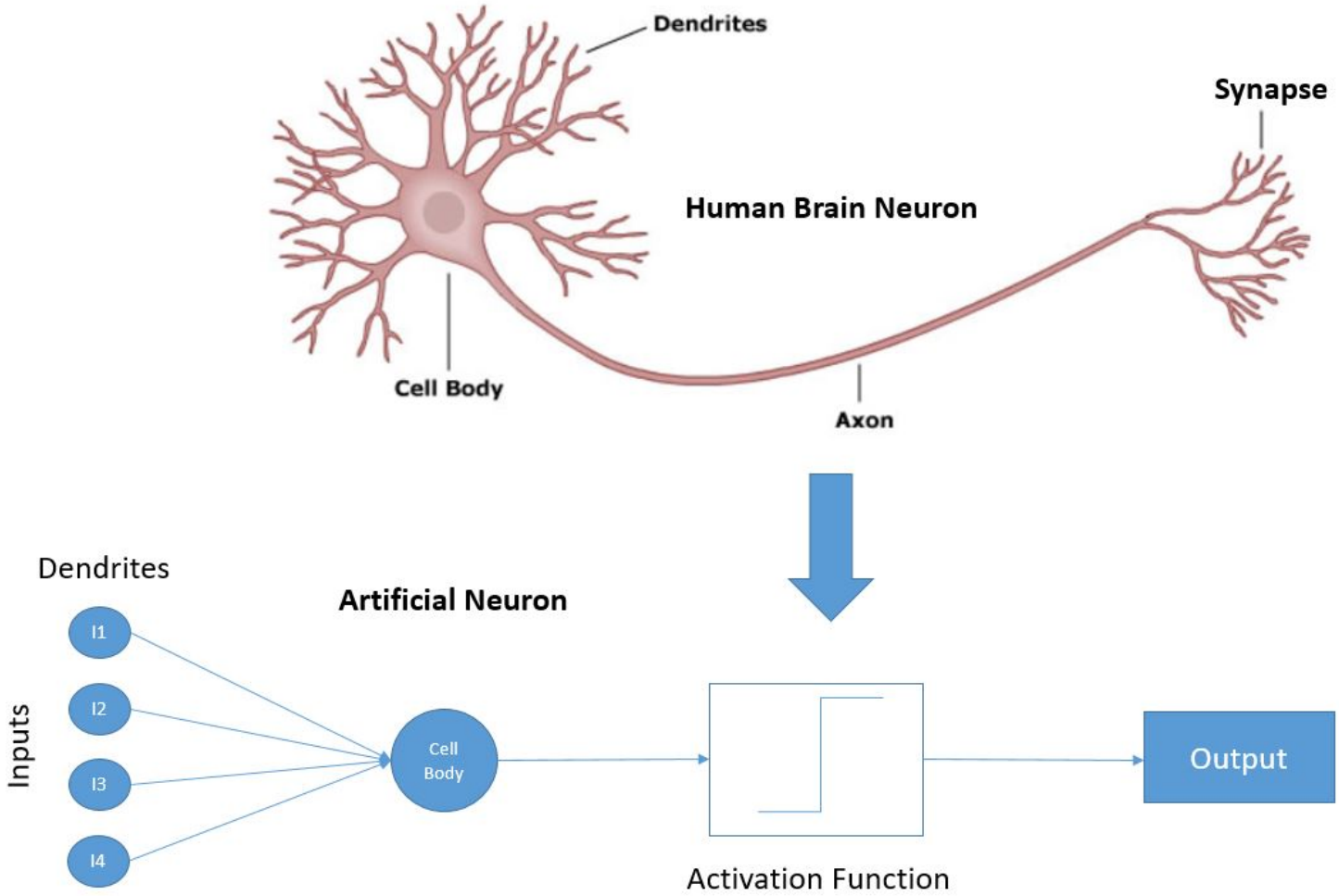
Contact: marcio.ferreira@uc.pt

Contents

- Perceptron
- Neural Networks
- Deep Neural Networks
- How to train DNN
  - Backpropagation algorithm
- Classification task with DNN

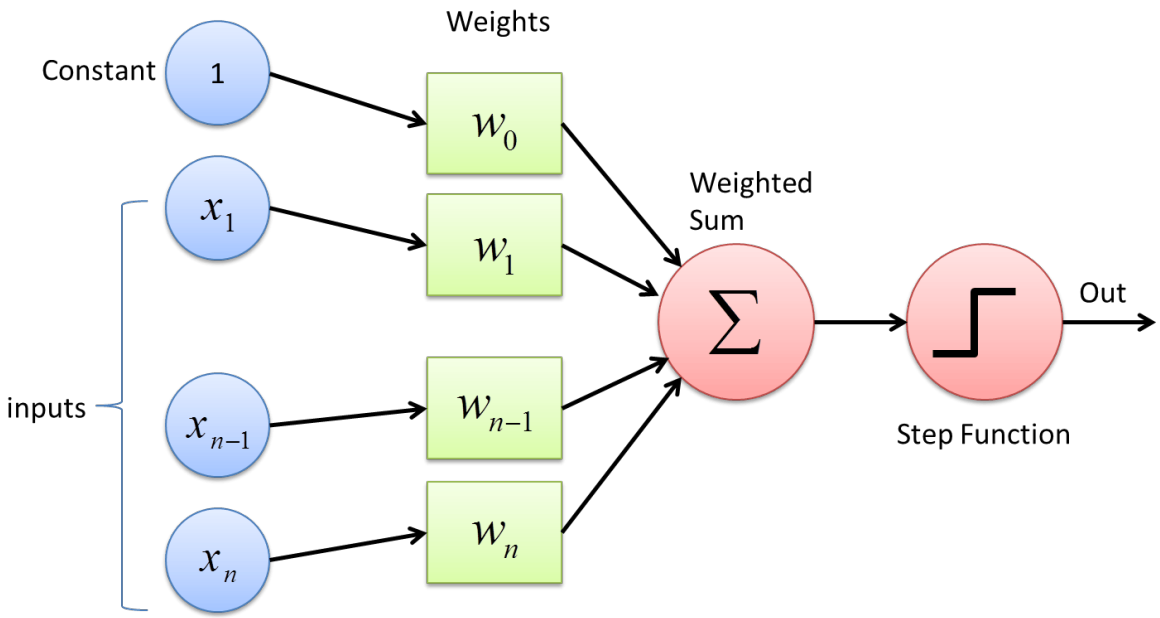
Biological Inspiration

- Biological neural network



Artificial neuron

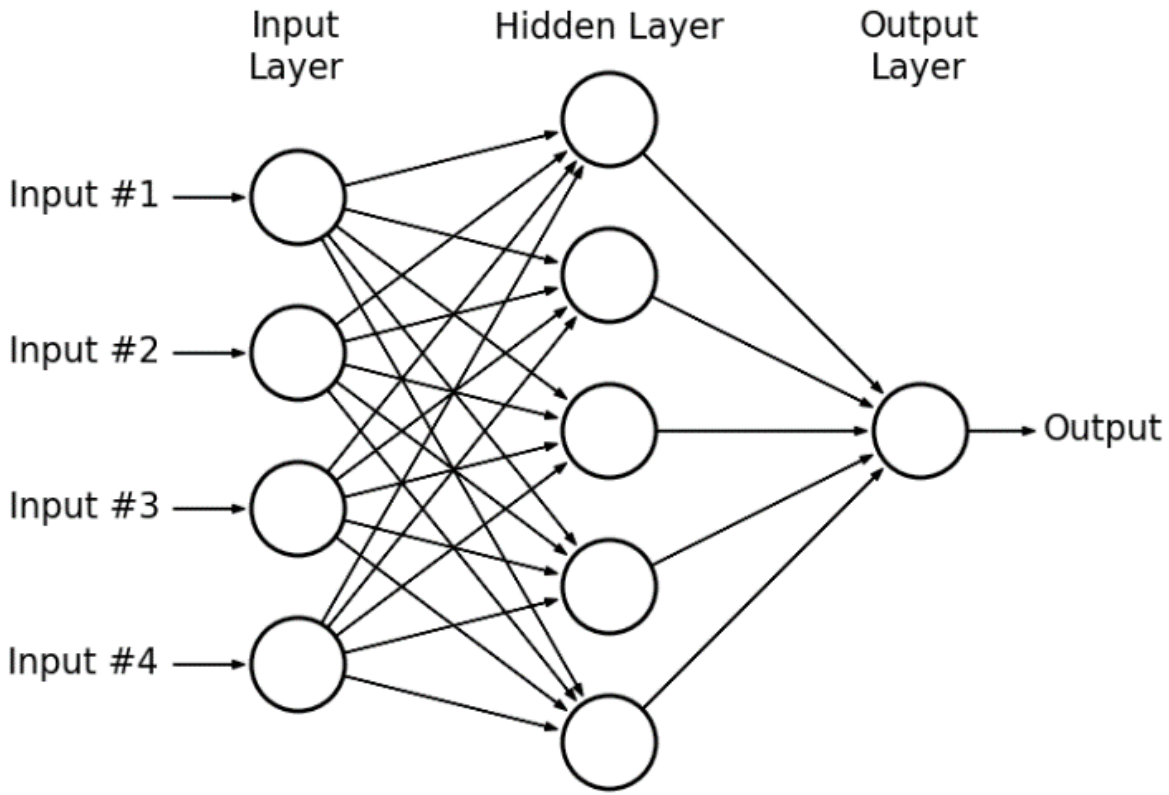
- Neural network as computing machines *(McCulloch-Pitts model)*
- Perceptrons are based on Linear Threshold Units (LTU)



- A single LTU can be used for simple linear binary classification

Perceptron

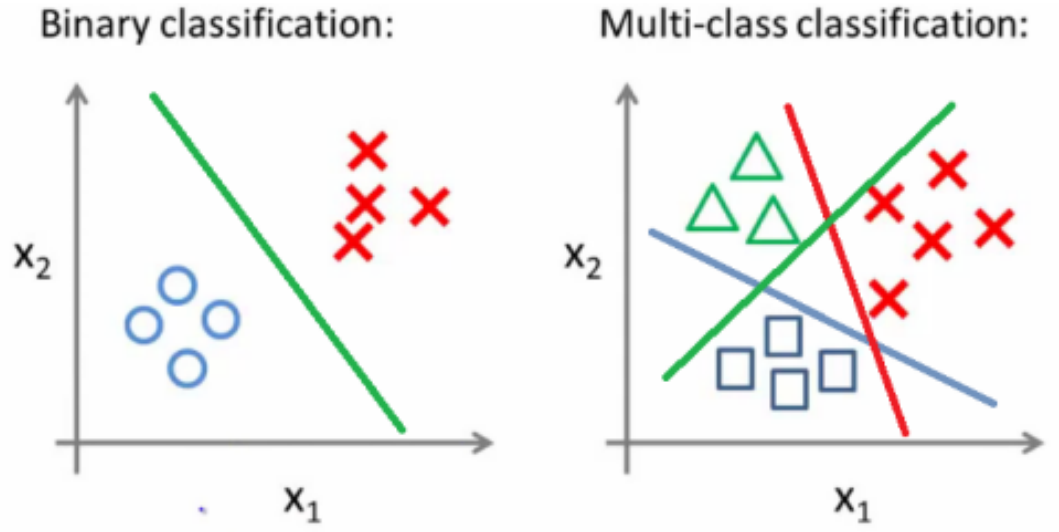
- A Perteptron is made of a single layer of LTUs
- The first type of artificial neural network (ANN)



- Multi-class **linear** classifier: the decision boundary of each neuron is linear

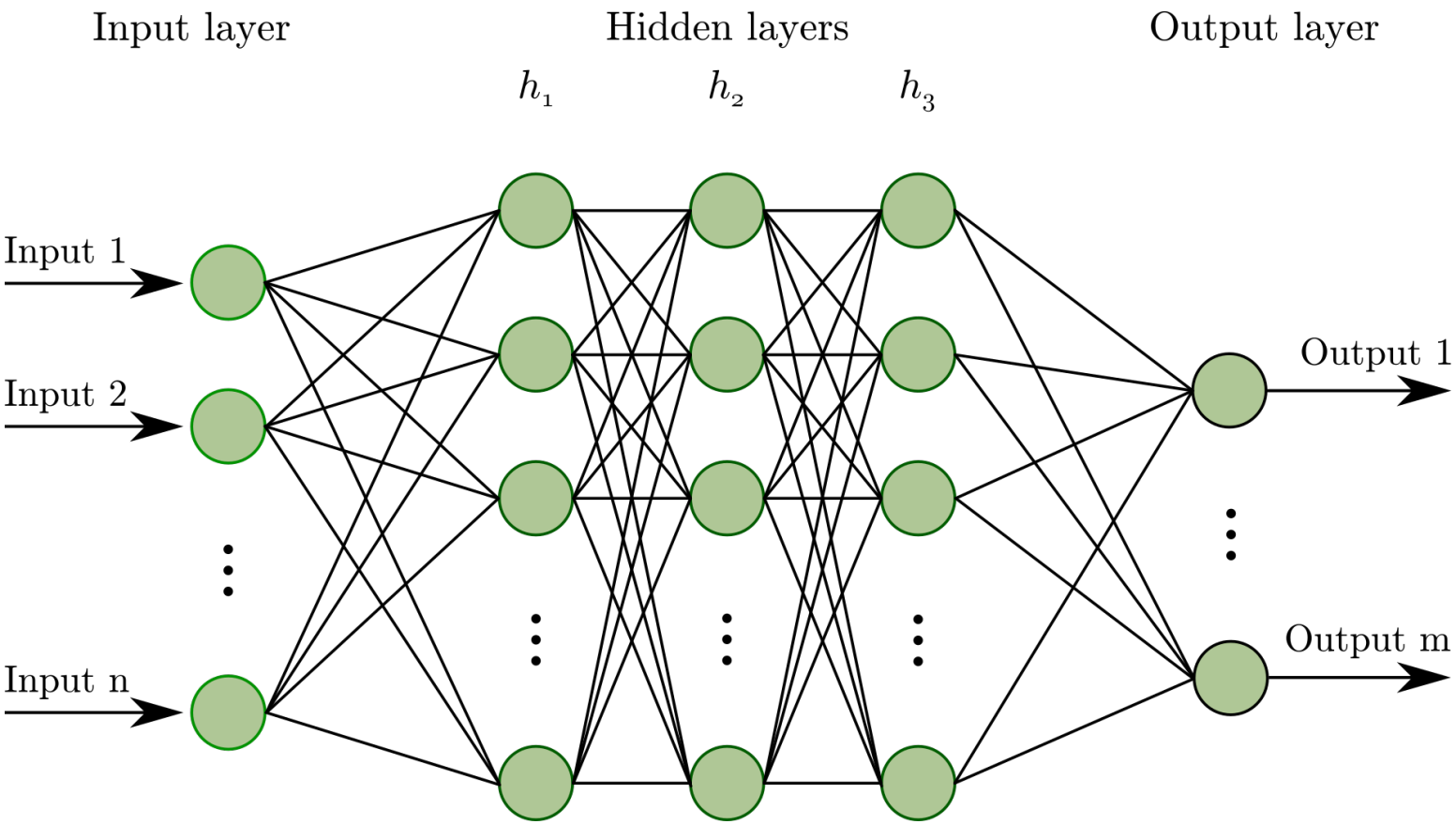
Perceptron

- Single-layer perceptrons only learn linearly separable patterns



Feed-forward Deep Neural Networks

- Multiple hidden layers increase the representational power of NN

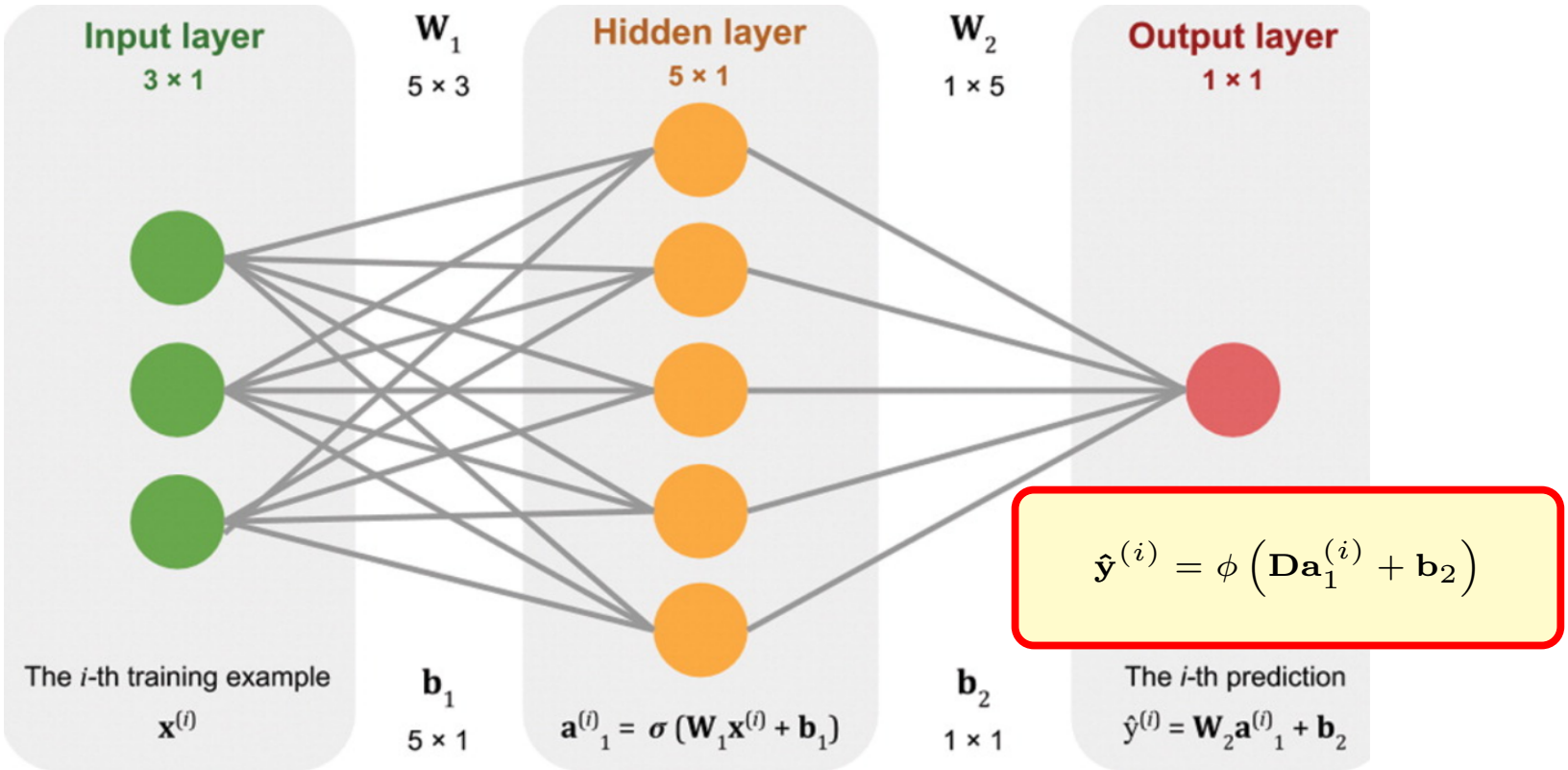


The math of Neural Networks

Input layer: $\mathbf{x}^{(i)}$

Hidden layer: $\mathbf{a}_1^{(i)} = \sigma\left(\mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}_1\right)$

Output layer: $\hat{\mathbf{y}}^{(i)} = \phi\left(\mathbf{D}\mathbf{a}_1^{(i)} + \mathbf{b}_2\right)$



[https://doi.org/10.1190/tle37080616.1]

The math of Neural Networks

Hidden layer: $\mathbf{a}_1^{(i)} = \sigma\left(\mathbf{\Sigma}_1^{(i)}\right)$, where $\mathbf{\Sigma}_1^{(i)} = \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}_1$

$$\begin{bmatrix} a_{11}^{(i)} \\ a_{12}^{(i)} \\ a_{13}^{(i)} \\ a_{14}^{(i)} \\ a_{15}^{(i)} \end{bmatrix} = \sigma\left(\begin{bmatrix} \Sigma_{11}^{(i)} \\ \Sigma_{12}^{(i)} \\ \Sigma_{13}^{(i)} \\ \Sigma_{14}^{(i)} \\ \Sigma_{15}^{(i)} \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{22} & W_{22} & W_{23} \\ W_{33} & W_{32} & W_{33} \\ W_{44} & W_{42} & W_{43} \\ W_{55} & W_{52} & W_{53} \end{bmatrix} \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ x_3^{(i)} \end{bmatrix} + \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \\ b_{14} \\ b_{15} \end{bmatrix}\right)$$

Output layer: $\hat{\mathbf{y}}^{(i)} = \phi\left(\mathbf{\Sigma}_2^{(i)}\right)$, where $\mathbf{\Sigma}_2^{(i)} = \mathbf{D}\mathbf{a}_1^{(i)} + \mathbf{b}_2$

$$\begin{bmatrix} \hat{y}_1^{(i)} \end{bmatrix} = \phi\left(\begin{bmatrix} \Sigma_{21}^{(i)} \end{bmatrix}\right) = \phi\left(\begin{bmatrix} D_{11} & D_{12} & D_{13} & D_{15} & D_{15} \end{bmatrix} \begin{bmatrix} a_{11}^{(i)} \\ a_{12}^{(i)} \\ a_{13}^{(i)} \\ a_{14}^{(i)} \\ a_{15}^{(i)} \end{bmatrix} + \begin{bmatrix} b_{21} \end{bmatrix}\right)$$

The math of Neural Networks

Normally, instead of single points, mini-batches of N points are fed into NN

Mini-batch: $$\begin{bmatrix} \mathbf{x}^{(1)} & \dots & \mathbf{x}^{(N)} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^{(1)} & \dots & \mathbf{x}_1^{(N)} \\ \mathbf{x}_2^{(1)} & \dots & \mathbf{x}_2^{(N)} \\ \mathbf{x}_3^{(1)} & \dots & \mathbf{x}_3^{(N)} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{a}_{11}^{(1)} & \dots & \mathbf{a}_{11}^{(N)} \\ \mathbf{a}_{12}^{(1)} & \dots & \mathbf{a}_{12}^{(N)} \\ \mathbf{a}_{13}^{(1)} & \dots & \mathbf{a}_{13}^{(N)} \\ \mathbf{a}_{14}^{(1)} & \dots & \mathbf{a}_{14}^{(N)} \\ \mathbf{a}_{15}^{(1)} & \dots & \mathbf{a}_{15}^{(N)} \end{bmatrix} = \sigma \left(\begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{22} & W_{22} & W_{23} \\ W_{33} & W_{32} & W_{33} \\ W_{44} & W_{42} & W_{43} \\ W_{55} & W_{52} & W_{53} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1^{(1)} & \dots & \mathbf{x}_1^{(N)} \\ \mathbf{x}_2^{(1)} & \dots & \mathbf{x}_2^{(N)} \\ \mathbf{x}_3^{(1)} & \dots & \mathbf{x}_3^{(N)} \end{bmatrix} + \begin{bmatrix} b_{11} & \dots & b_{11} \\ b_{12} & \dots & b_{12} \\ b_{13} & \dots & b_{13} \\ b_{14} & \dots & b_{14} \\ b_{15} & \dots & b_{15} \end{bmatrix} \right)$$

$$\begin{bmatrix} \hat{\mathbf{y}}^{(1)} & \dots & \hat{\mathbf{y}}^{(N)} \end{bmatrix} = \phi \left(\begin{bmatrix} D_{11} & D_{12} & D_{13} & D_{15} & D_{15} \end{bmatrix} \begin{bmatrix} \mathbf{a}_{11}^{(1)} & \dots & \mathbf{a}_{11}^{(N)} \\ \mathbf{a}_{12}^{(1)} & \dots & \mathbf{a}_{12}^{(N)} \\ \mathbf{a}_{13}^{(1)} & \dots & \mathbf{a}_{13}^{(N)} \\ \mathbf{a}_{14}^{(1)} & \dots & \mathbf{a}_{14}^{(N)} \\ \mathbf{a}_{15}^{(1)} & \dots & \mathbf{a}_{15}^{(N)} \end{bmatrix} + \begin{bmatrix} b_{21} & \dots & b_{21} \end{bmatrix} \right)$$

- **The output of a mini-batch is obtained by matrix operations**

Training a NN

- Training the model consists in adjusting the network' weights to minimize a cost function

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \mathcal{C}(\mathbf{x}, \mathbf{y}, \mathbf{w})$$

- Moving in the direction of negative and large $\nabla\mathcal{C}(\mathbf{x}, \mathbf{y}, \mathbf{w})$ (gradient descent): initialize $\mathbf{w}$ to some value $\mathbf{w}_0$ and update its value according to

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla_{\mathbf{w}} \mathcal{C}(\mathbf{w}_t)$$

- **To speed up calculations and add stochasticity** (low probability of getting stuck in local minimum), the gradients are approximated on a subset of data (**mini-batch**)

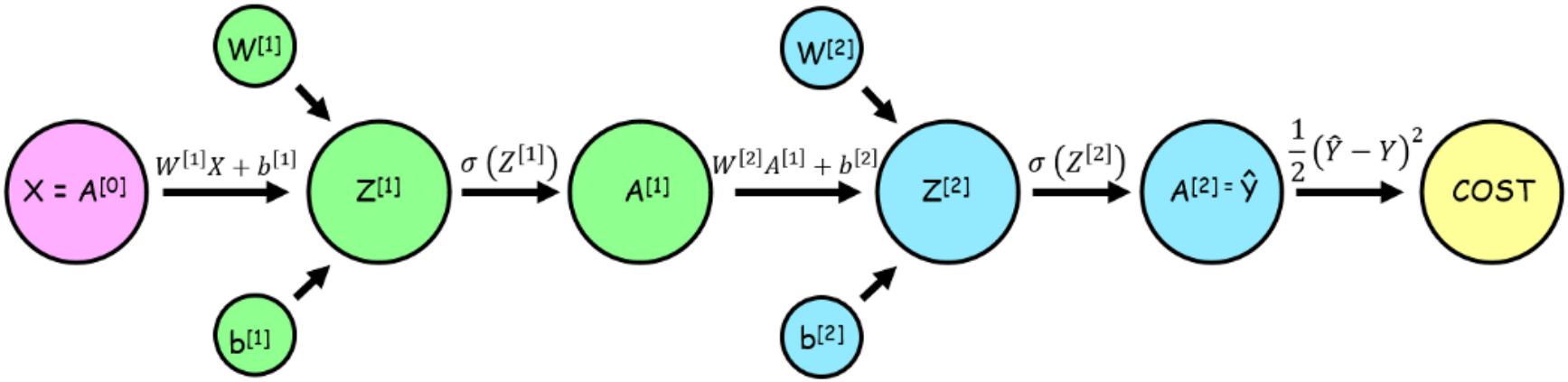
$$\nabla_{\mathbf{w}} \mathcal{C}(\mathbf{w}_t) \rightarrow \nabla_{\mathbf{w}} \mathcal{C}^{\mathsf{MB}}(\mathbf{w}) = \sum_{i \in B_k} \nabla_{\mathbf{w}} \mathcal{C}(\mathbf{x}_i, \mathbf{w})$$

Backpropagation algorithm

- For each training point (or mini-batch), a huge number of derivatives computations is required.
- **Backpropagation is an efficient algorithm that accomplishes this complex task**
  - Forward propagation step: data is passed through a network to determine the cost function
  - Backward propagation step: adjust the model's parameters to reduce the cost function

Forward propagation

- Computational graph for the forward pass



[https://towardsdatascience.com/neural-networks-backpropagation-by-dr-lihi-gur-arie-27be67d8fdce]

$$1) \quad \mathbf{Z}^{(1)} = \mathbf{W}^{(1)}\mathbf{X} + \mathbf{b}^{(1)}$$

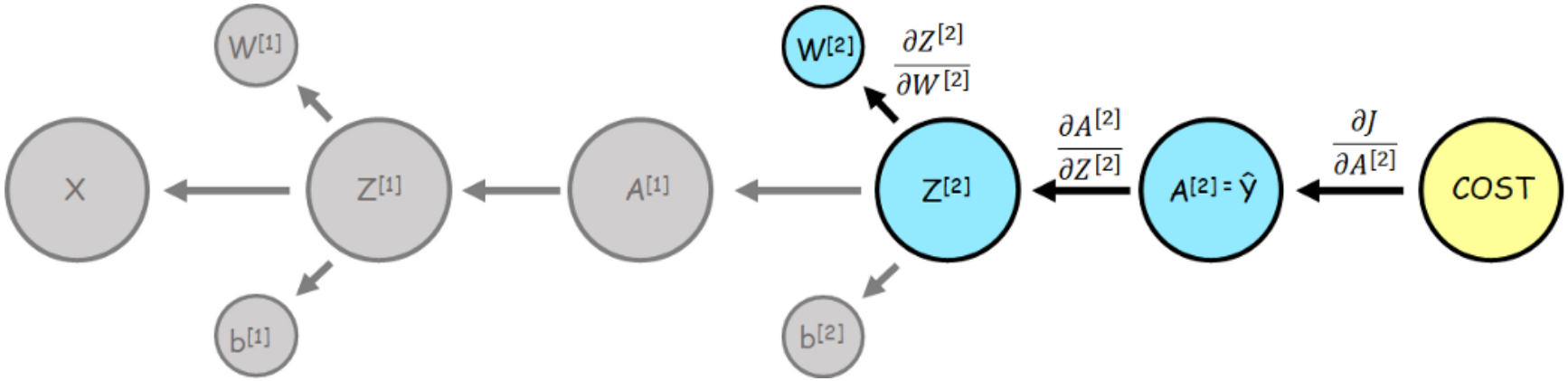
$$2) \quad \mathbf{A}^{(1)} = \sigma\left(\mathbf{Z}^{(1)}\right)$$

$$3) \quad \mathbf{Z}^{(2)} = \mathbf{W}^{(2)}\mathbf{A}^{(1)} + \mathbf{b}^{(2)}$$

$$4) \quad \hat{\mathbf{Y}} = \sigma\left(\mathbf{Z}^{(2)}\right)$$

$$5) \quad \mathcal{C} = \frac{1}{2}(\hat{\mathbf{Y}} - Y)^2 \rightarrow \mathcal{C}\left(\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \mathbf{W}^{(1)}, \mathbf{W}^{(2)}\right)$$

Backward propagation

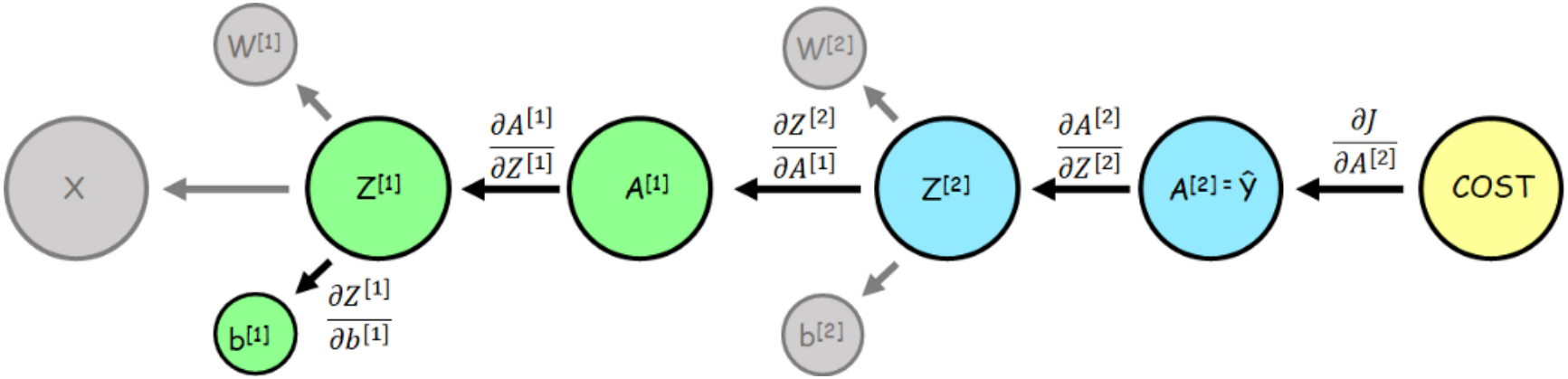


[https://towardsdatascience.com/neural-networks-backpropagation-by-dr-lihi-gur-arie-27be67d8fdce]

$$\frac{\partial \mathcal{C}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{C}}{\partial \mathbf{A}^{(2)}} \frac{\partial \mathbf{A}^{(2)}}{\partial \mathbf{Z}^{(2)}} \frac{\partial \mathbf{Z}^{(2)}}{\partial \mathbf{W}^{(2)}}$$

- The total derivative is determined from the product of local derivatives
- All information required to compute these local gradients was saved during the forward pass

Backward propagation



[https://towardsdatascience.com/neural-networks-backpropagation-by-dr-lihi-gur-arie-27be67d8fdce]

$$\frac{\partial \mathcal{C}}{\partial \mathbf{b}^{(1)}} = \frac{\partial \mathcal{C}}{\partial \mathbf{A}^{(2)}} \frac{\partial \mathbf{A}^{(2)}}{\partial \mathbf{Z}^{(2)}} \frac{\partial \mathbf{Z}^{(2)}}{\partial \mathbf{A}^{(1)}} \frac{\partial \mathbf{A}^{(1)}}{\partial \mathbf{Z}^{(1)}} \frac{\partial \mathbf{Z}^{(1)}}{\partial \mathbf{b}^{(1)}}$$

- After applying the procedure for each network's weight

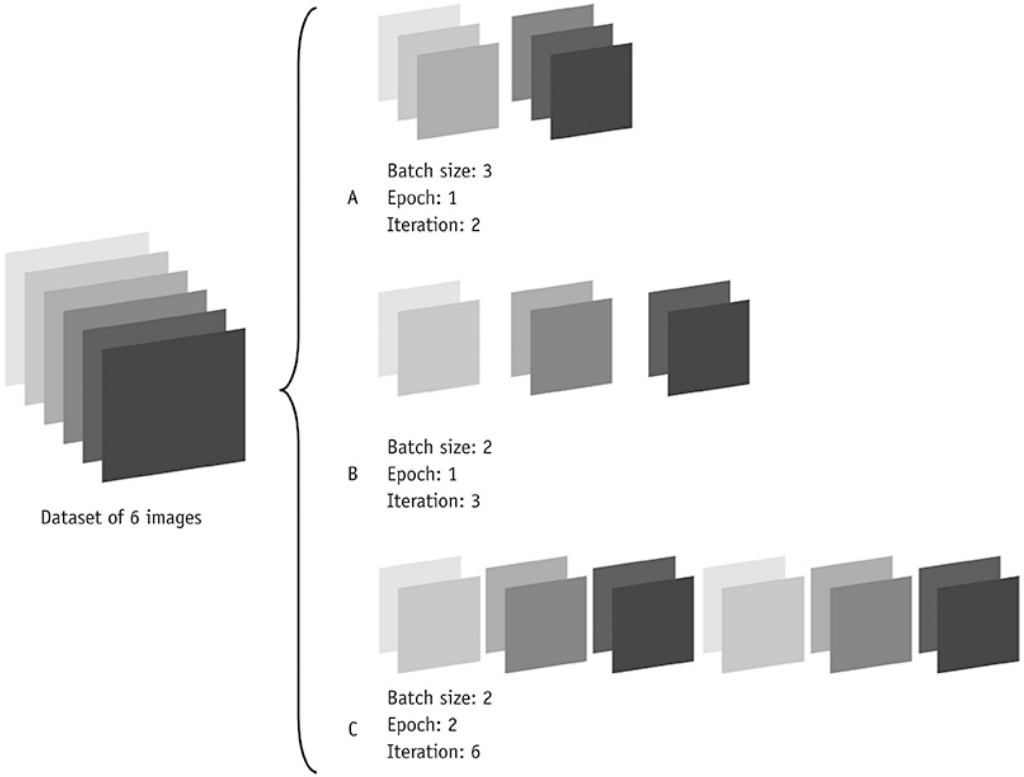
$$\begin{pmatrix} \mathbf{W}^{(1)}_{t+1} \\ \mathbf{W}^{(2)}_{t+1} \\ \mathbf{b}^{(1)}_{t+1} \\ \mathbf{b}^{(2)}_{t+1} \end{pmatrix} = \begin{pmatrix} \mathbf{W}^{(1)}_{t} \\ \mathbf{W}^{(2)}_{t} \\ \mathbf{b}^{(1)}_{t} \\ \mathbf{b}^{(2)}_{t} \end{pmatrix} - \eta_t \begin{pmatrix} \partial \mathcal{C}/\partial \mathbf{W}^{(1)}_{t} \\ \partial \mathcal{C}/\partial \mathbf{W}^{(2)}_{t} \\ \partial \mathcal{C}/\partial \mathbf{b}^{(1)}_{t} \\ \partial \mathcal{C}/\partial \mathbf{b}^{(2)}_{t} \end{pmatrix}$$

Notation: batch size, mini-batch, epoch, and iteration

- Consider a dataset with $N = 100$ points
- By setting a **batch size** of 20, we are creating 5 **mini-batches**:

$$\{B_1, B_2, B_3, B_4, B_5\}$$

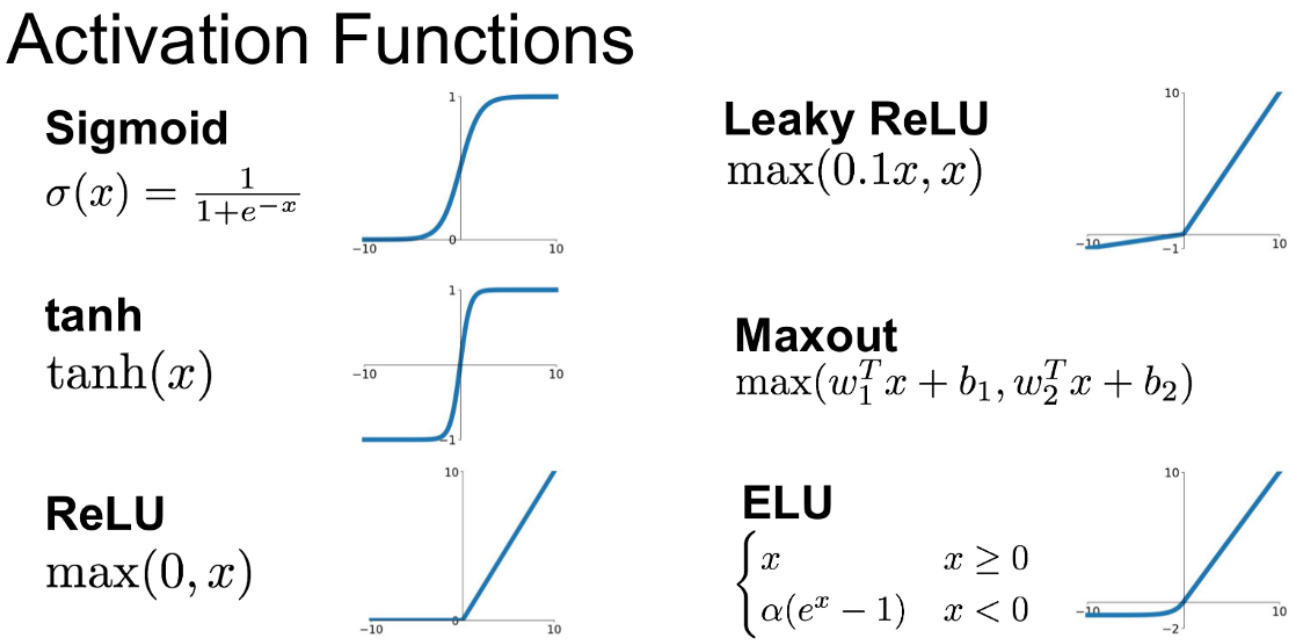
- An **epoch** refers to one cycle through the full dataset
- Having 5 mini-batches, an epoch is completed in 5 **iterations**.



[https://doi.org/10.3348/kjr.2019.0312]

Activation functions

- There are different activation functions with specific computational properties
- Their derivatives' properties have a crucial impact on training NN
  - Sigmoid/tanh have vanishing gradients for $|x| \gg 1$



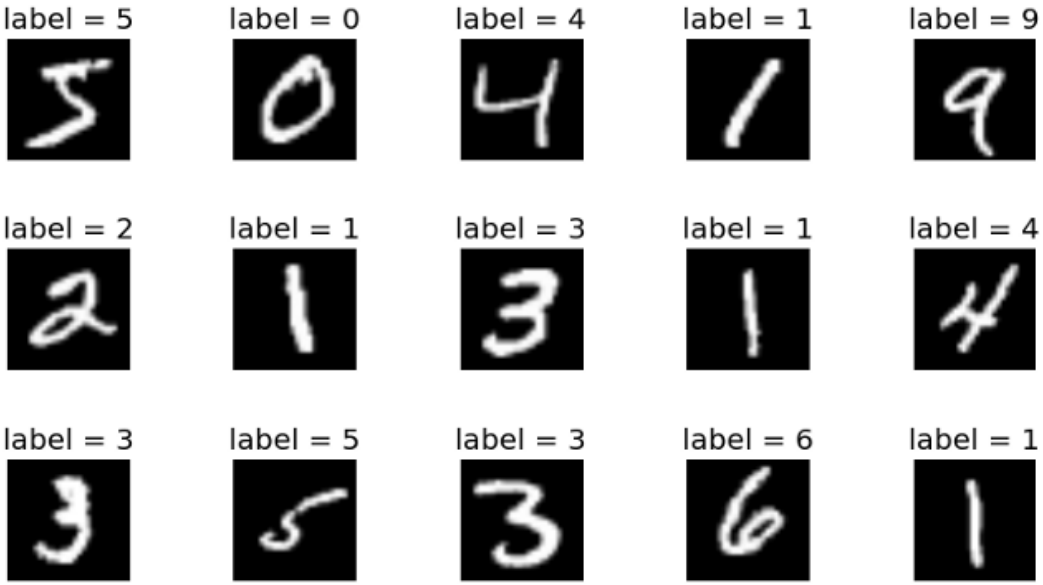
[https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092]

Using DNN: classification problem

- MNIST dataset

$$\mathcal{D}\left\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_N, y_N)\right\},$$

where $\mathbf{x}_i$ is a 2D array of gray-scale and y is the label



- Training set: 60000 images
- Testing set: 10000 images

MNIST dataset

- The whole dataset is represented by a 3D array $(N \times X \times Y)$

$$x_{ijk} \in X_{\text{train}} \rightarrow [1:60000, 1:28, 1:28] \text{ (rank 3 tensor)}$$

- 2D arrays (slices) represent samples $(X \times Y)$

$$x_{1jk} \rightarrow [1, 1:28, 1:28] \text{ (rank 2 tensor)}$$

$$x_{5jk} \rightarrow [5, 1:28, 1:28]$$

- The outut values are in a 1D array (N)

$$y_{\text{train}} = \{y_1, y_2, ..., y_{60000}\} \rightarrow [1:60000] \text{ (rank 1 tensor)}$$

- Their values indicate the class:

$$y_1 = [1] = 5, \quad y_2 = [2] = 0, \quad y_3 = [3] = 4, \quad ...$$

- y_i are *slices* (scalars, rank 0 tensors) from the vector y_{train}

A sample from the dataset

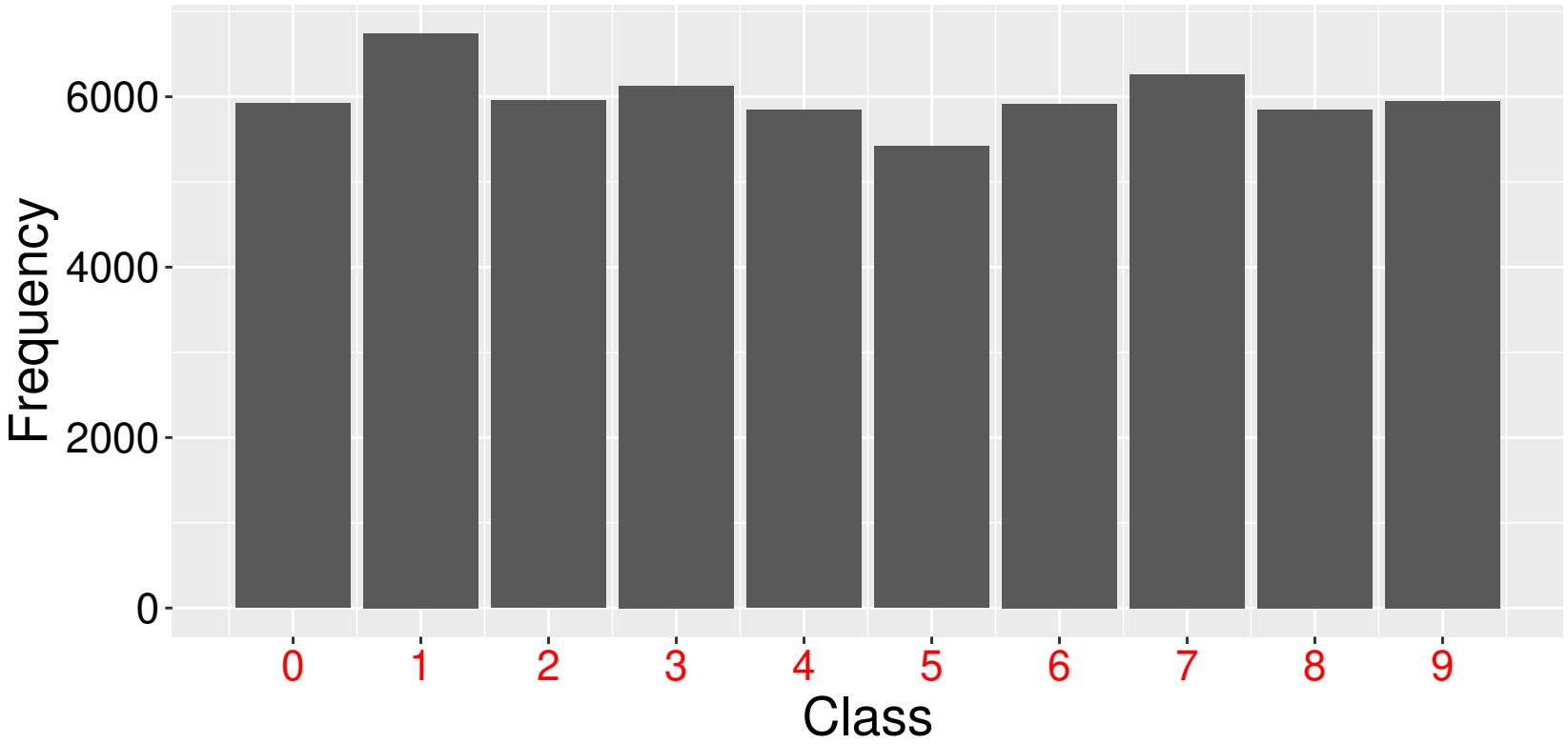
- $x_{1jk} \rightarrow [28 \times 28]$ (2D array with the gray-scale of each pixel)

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22	V23	V24	V25	V26	V27	V28
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
2	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
3	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
4	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
5	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
6	0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0	0
7	0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0	0
8	0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0	0
9	0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	14	1	154	253	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	249	253	249	64	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	136	253	253	253	212	135	132	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	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
27	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
28	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

- The output space: $y_1 = 5$

Balanced training set

- We need to check whether the training set is balanced
- The number of observations per class shows small deviations



Transform data

- We need to reshape the input space

$$x_{ij} \in X_{\text{train}} \rightarrow [1 : 60000, 784]$$

- Each imagine is now represented by a vector of length 784
- Rescale the dataset (normalize the features)

$$x_{ij} \rightarrow x_{ij}/255$$

- Apply the one-hot encoding to the classes

$$y = 0 \rightarrow \mathbf{y} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$y = 1 \rightarrow \mathbf{y} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\dots$$

$$y = 9 \rightarrow \mathbf{y} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

One-hot encoding

- Our initial y variables contained label values $\{0, 1, ..., 9\}$ rather than numeric values
- One-hot encoding ensures that our model does not consider higher numbers to be more important

	V1
1:	5
2:	0
3:	4
4:	1
5:	9

59996:	8
59997:	3
59998:	5
59999:	6
60000:	8

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
1:	0	0	0	0	0	1	0	0	0	0
2:	1	0	0	0	0	0	0	0	0	0
3:	0	0	0	0	1	0	0	0	0	0
4:	0	1	0	0	0	0	0	0	0	0
5:	0	0	0	0	0	0	0	0	0	1

59996:	0	0	0	0	0	0	0	0	1	0
59997:	0	0	0	1	0	0	0	0	0	0
59998:	0	0	0	0	0	1	0	0	0	0
59999:	0	0	0	0	0	0	1	0	0	0
60000:	0	0	0	0	0	0	0	0	1	0

- $y_{ij} \in y_{\text{train}} \rightarrow [1 : 60000, 10]$ (2D array, rank 2 tensor)

Define the NN model (using Keras in R)

```
inputs <- layer_input(shape = c(784))

predictions <- inputs %>%
  layer_dense(units = 256, activation = 'relu') %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')

model <- keras_model(inputs = inputs, outputs = predictions)
```

```
_________________________________________________________________________________________________
Layer (type)                                 Output Shape                              Param #
=================================================================================================
input   (InputLayer)                         [(None, 784)]                             0
_________________________________________________________________________________________________
dense   (Dense)                              (None, 256)                               200960
_________________________________________________________________________________________________
dense   (Dense)                              (None, 128)                               32896
_________________________________________________________________________________________________
dense   (Dense)                              (None, 10)                                1290
=================================================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
```

- Softmax function: $P(y = j \mid \mathbf{x}) = \dfrac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$

Network output: $(P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9)$

Setting a batch size

- Mini-batches of size 128

```
inputs <- layer_input(batch_shape = c(128,784))

predictions <- inputs %>%
  layer_dense(units = 256, activation = 'relu') %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')

model <- keras_model(inputs = inputs, outputs = predictions)
```

```
_____________________________________________________________________________________________________
Layer (type)                                 Output Shape                              Param #
=====================================================================================================
input_4 (InputLayer)                         [(128, 784)]                              0
_____________________________________________________________________________________________________
dense_23 (Dense)                             (128, 256)                                200960
_____________________________________________________________________________________________________
dense_22 (Dense)                             (128, 128)                                32896
_____________________________________________________________________________________________________
dense_21 (Dense)                             (128, 10)                                 1290
=====================================================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
```

Compile the NN model

- Categorical crossentropy (loss function): how distinguishable two discrete probability distributions are

$$L(\mathbf{W}) = -\sum_{i=0}^{9} y_i \cdot \log \hat{y}_i(\mathbf{W})$$

  - $\mathbf{W}$ represents all the (235146) NN weights
- Adam Optimizer [several available: SGD, RMSprop,...]
- Metric: the fraction of the images correctly classified (accuracy)

```
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = 'adam',
  metrics = c("accuracy")
)
```

Training the NN model

- Randomly split the train dataset (60000 images):
  - 80% for training and 20% for validation
- 30 epochs: number of times the entire training set (48000 images) will pass through the NN
- Mini-batches of size of 128:

$$X_{in} \rightarrow \{[1:128, 784], [129:257, 784], ...\}$$

- Each epoch has $48000/128 = 375$ iterations
  - number of times the $\mathbf{W}$ are updated in each epoch

```
model %>% fit(
  x_train, y_train,
  epochs = 30,
  batch_size = 128,
  validation_split = 0.2
)
```

Output of the training stage

```
Epoch 1/30
375/375 [==============================] - 12s 30ms/step - loss: 0.5482 - accuracy: 0.8456 - val_loss: 0.1427 - val_accuracy: 0.9603
Epoch 2/30
375/375 [==============================] - 7s 19ms/step - loss: 0.1192 - accuracy: 0.9650 - val_loss: 0.1060 - val_accuracy: 0.9670
Epoch 3/30
375/375 [==============================] - 14s 36ms/step - loss: 0.0762 - accuracy: 0.9769 - val_loss: 0.0879 - val_accuracy: 0.9728
Epoch 4/30
375/375 [==============================] - 15s 40ms/step - loss: 0.0514 - accuracy: 0.9847 - val_loss: 0.0859 - val_accuracy: 0.9746
Epoch 5/30
375/375 [==============================] - 14s 38ms/step - loss: 0.0353 - accuracy: 0.9899 - val_loss: 0.0885 - val_accuracy: 0.9741
Epoch 6/30
375/375 [==============================] - 10s 27ms/step - loss: 0.0274 - accuracy: 0.9920 - val_loss: 0.0816 - val_accuracy: 0.9774
```

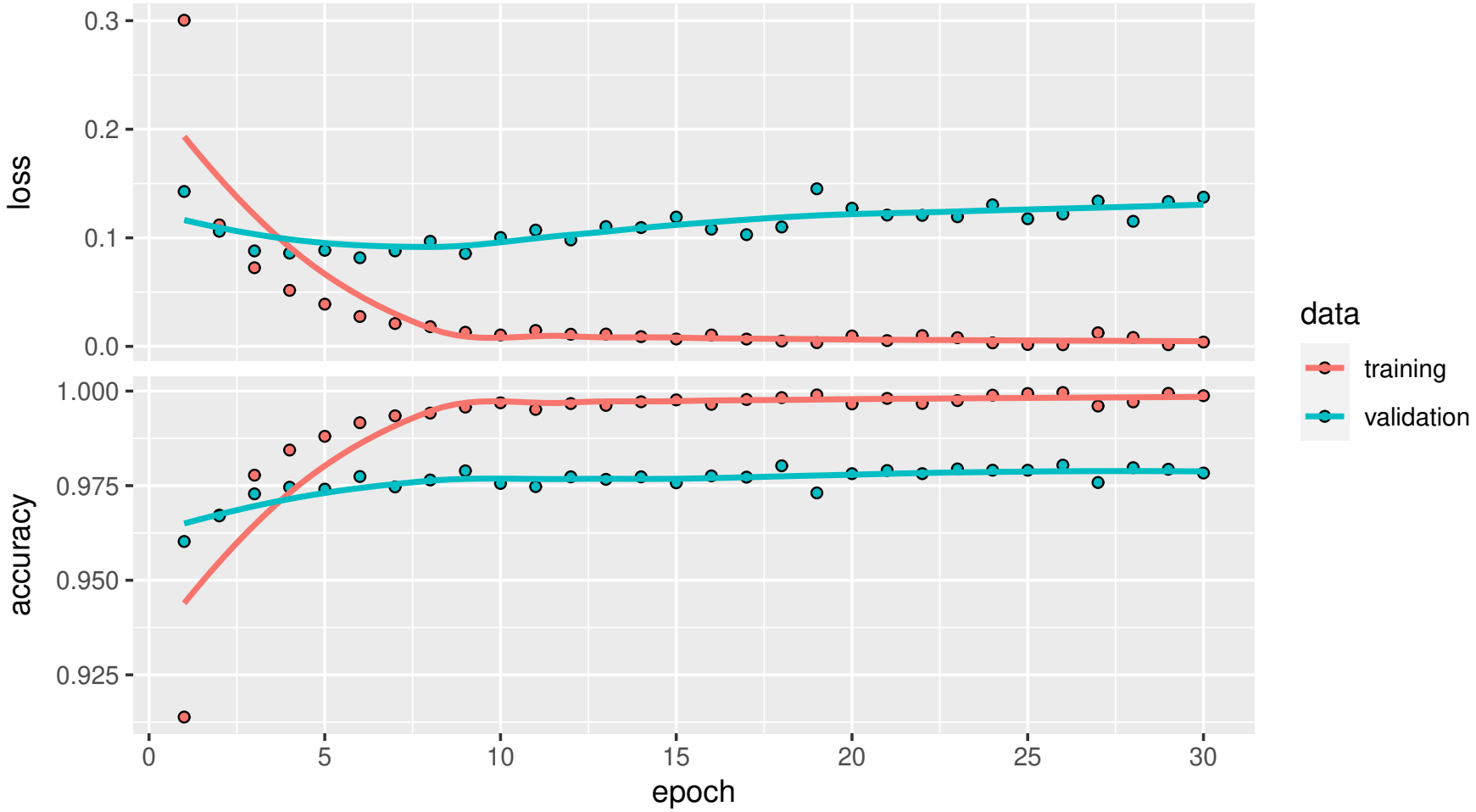
■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

```
Epoch 25/30
375/375 [==============================] - 13s 36ms/step - loss: 0.0017 - accuracy: 0.9993 - val_loss: 0.1176 - val_accuracy: 0.9791
Epoch 26/30
375/375 [==============================] - 14s 37ms/step - loss: 0.0011 - accuracy: 0.9997 - val_loss: 0.1219 - val_accuracy: 0.9804
Epoch 27/30
375/375 [==============================] - 13s 35ms/step - loss: 0.0070 - accuracy: 0.9978 - val_loss: 0.1339 - val_accuracy: 0.9758
Epoch 28/30
375/375 [==============================] - 10s 26ms/step - loss: 0.0099 - accuracy: 0.9967 - val_loss: 0.1152 - val_accuracy: 0.9797
Epoch 29/30
375/375 [==============================] - 15s 40ms/step - loss: 0.0014 - accuracy: 0.9994 - val_loss: 0.1334 - val_accuracy: 0.9793
Epoch 30/30
375/375 [==============================] - 14s 39ms/step - loss: 0.0040 - accuracy: 0.9988 - val_loss: 0.1375 - val_accuracy: 0.9783
```

- **Do we understand the meaning of all these numbers?**

Deep Neural Networks with keras

- Graphical representation of the learning stage



Out-of-sample performance of the model

- The final model's performance is measured on the test set
  - 10000 images

```
> model %>% evaluate(x_test, y_test, verbose=2)
313/313 - 3s - loss: 0.1250 - accuracy: 0.9800
     loss  accuracy
0.1249673 0.9800000
```

- Accuracy of 98% (the model correctly classifies 98 out of 100 figures)
- The *evaluate()* function uses batches (default size of 32) just to speed-up evaluation (10000/32=312.5)

Model output

- Model predictions
- Last layer: $(P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9)$

	0	1	2	3	4	5	6	7	8	9
1:	2.430218e-15	6.946803e-15	7.204404e-12	1.802491e-08	2.795768e-21	1.874775e-14	3.202079e-20	9.999995e-01	6.316579e-14	5.257895e-07
2:	1.330668e-18	2.885281e-14	1.000000e+00	1.690751e-16	6.226776e-28	4.065791e-20	9.953934e-19	5.430409e-27	3.760663e-14	1.003303e-28
3:	3.368088e-12	9.999995e-01	4.782248e-07	3.428024e-13	4.998332e-10	8.679319e-12	3.595326e-10	2.322528e-08	4.713738e-08	2.584044e-15
4:	1.000000e+00	4.121874e-18	1.144198e-12	2.128090e-18	1.623171e-17	3.448859e-18	1.688180e-12	6.584955e-17	1.671761e-16	2.085538e-16
5:	9.358533e-14	2.531308e-14	4.966997e-12	1.619905e-17	1.000000e+00	2.026449e-20	9.402332e-14	7.990271e-14	3.389842e-15	1.917231e-09

9996:	1.840926e-28	1.223570e-18	1.000000e+00	4.778362e-16	8.188086e-38	1.464597e-23	2.489068e-28	2.145684e-15	7.508420e-16	2.860889e-30
9997:	1.321077e-19	3.057360e-18	4.561980e-12	9.999996e-01	2.132357e-24	1.630280e-08	4.081237e-27	2.533833e-15	5.045037e-13	3.256052e-07
9998:	2.140457e-25	5.025506e-20	3.493071e-24	6.937665e-24	1.000000e+00	4.605477e-27	2.479116e-22	7.837477e-17	2.080704e-19	9.053961e-13
9999:	8.241506e-25	1.659762e-26	8.974243e-23	1.003467e-20	1.385951e-28	1.000000e+00	6.783952e-20	1.148829e-18	5.537012e-14	3.024205e-21
10000:	3.608359e-22	1.468738e-29	1.798328e-22	1.007004e-26	2.697474e-18	8.694002e-20	1.000000e+00	6.897178e-34	3.813170e-19	4.570940e-29

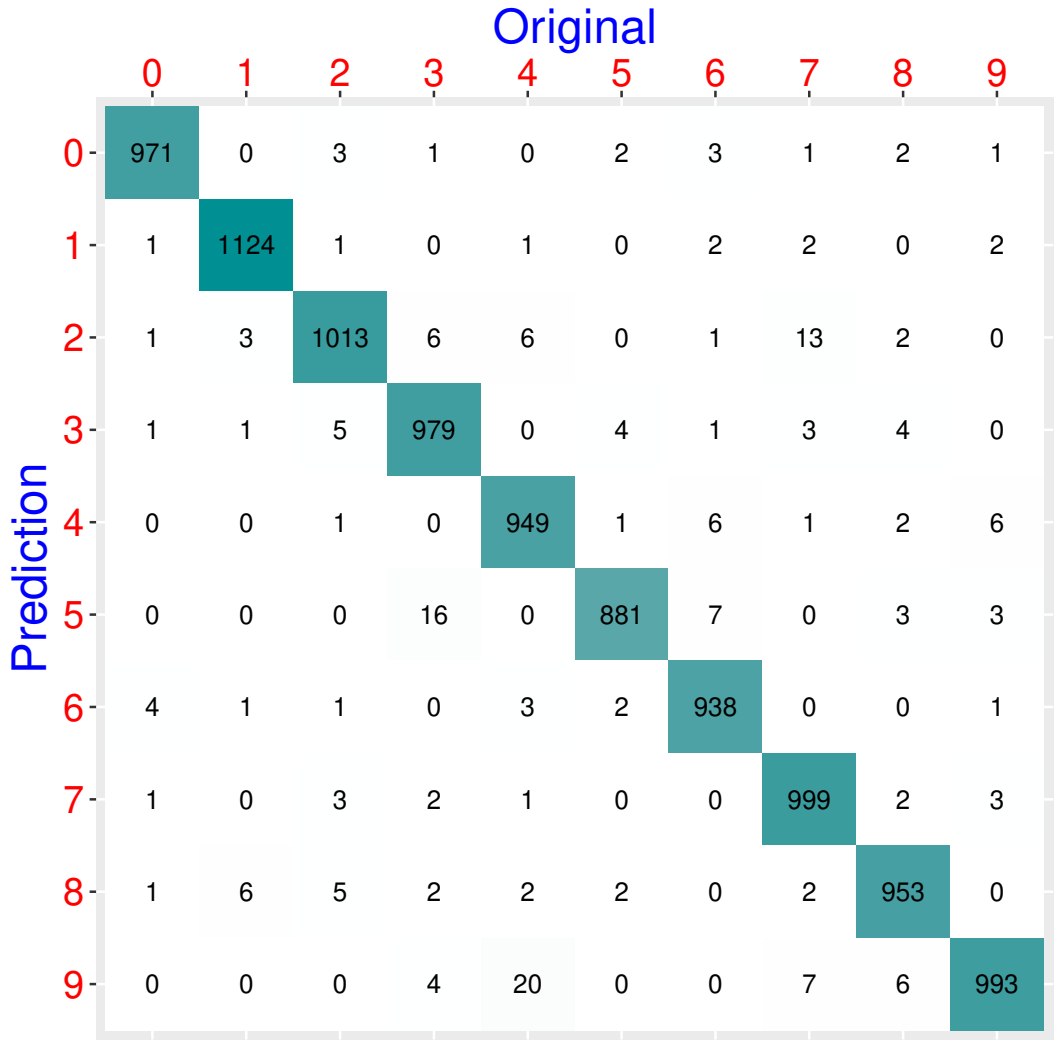
- Dataset values

	0	1	2	3	4	5	6	7	8	9
1:	0	0	0	0	0	0	0	1	0	0
2:	0	0	1	0	0	0	0	0	0	0
3:	0	1	0	0	0	0	0	0	0	0
4:	1	0	0	0	0	0	0	0	0	0
5:	0	0	0	0	1	0	0	0	0	0

9996:	0	0	1	0	0	0	0	0	0	0
9997:	0	0	0	1	0	0	0	0	0	0
9998:	0	0	0	0	1	0	0	0	0	0
9999:	0	0	0	0	0	1	0	0	0	0
10000:	0	0	0	0	0	0	1	0	0	0

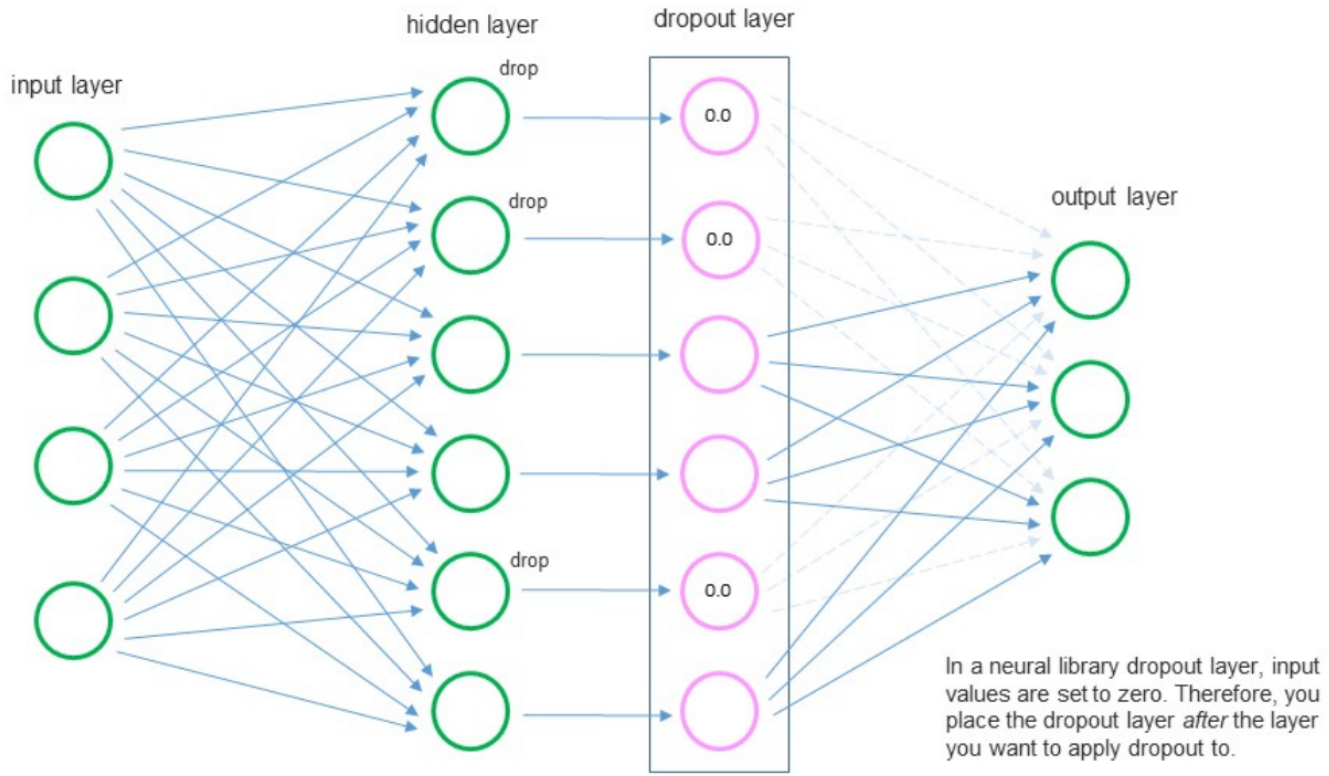
Confusion Matrix

- Summary of prediction results



Regularization for NN: dropout

- The dropout layer **prevents overfitting** by randomly assigning neurons to 0 at each step during training with a given frequency.
- dropout rate $= 0.5$: half of the neurons excluded from each update iteraction.



https://jamesmccaffrey.wordpress.com/2018/05/11/neural-network-library-dropout-layers/

Adding dropout to the model

- Adding dropout layers

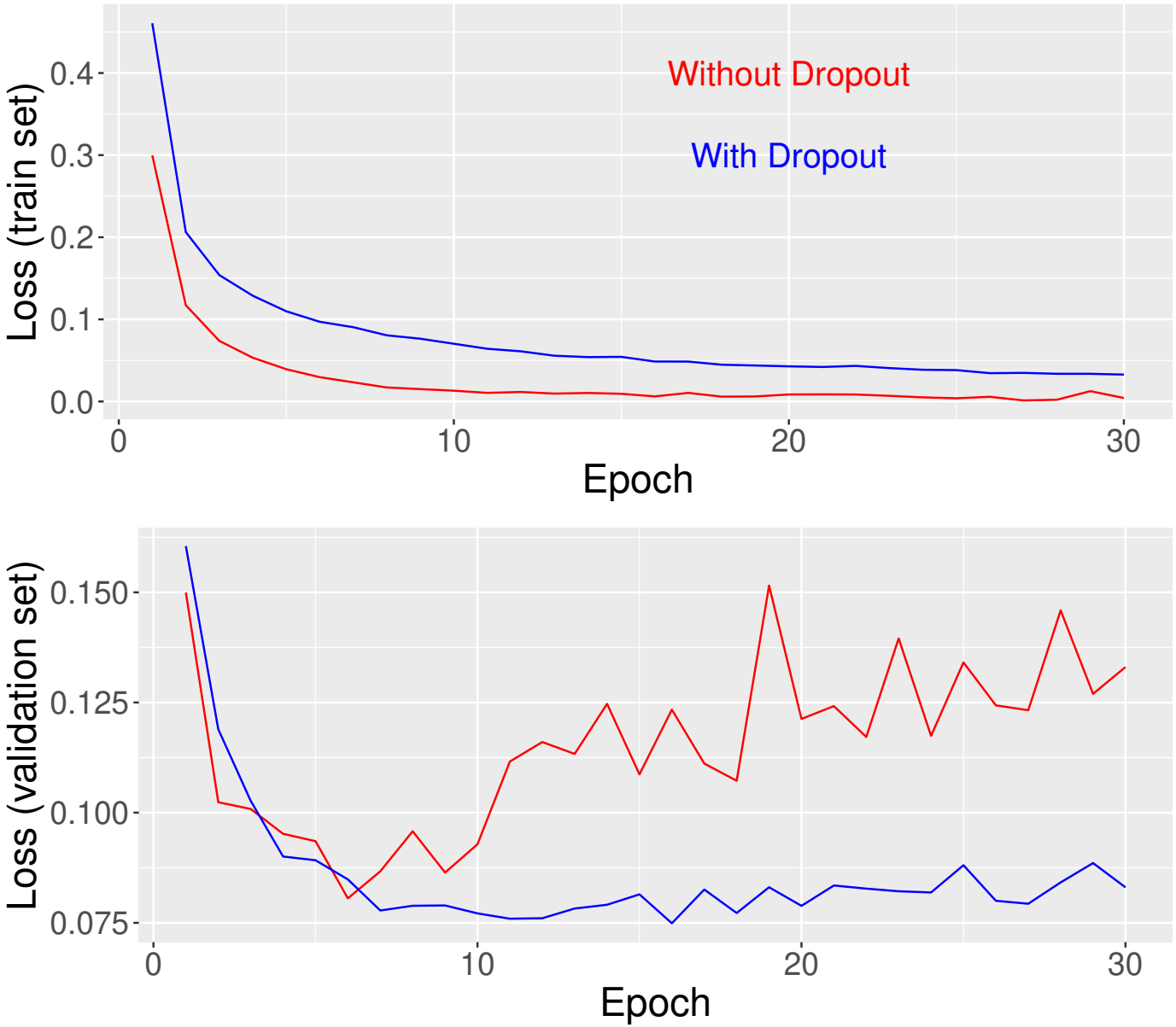
```
inputs_m2 <- layer_input(batch_shape = c(128,784))

predictions_m2 <- inputs_m2 %>%
  layer_dense(units = 256, activation = 'relu') %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = 'softmax')

model_m2 <- keras_model(inputs = inputs_m2, outputs = predictions_m2)
```

```
_____________________________________________________________________________________________________________
Layer (type)                                    Output Shape                                    Param #
=============================================================================================================
input   (InputLayer)                            [(128, 784)]                                    0
_____________________________________________________________________________________________________________
dense   (Dense)                                 (128, 256)                                      200960
_____________________________________________________________________________________________________________
dropout   (Dropout)                             (128, 256)                                      0
_____________________________________________________________________________________________________________
dense   (Dense)                                 (128, 128)                                      32896
_____________________________________________________________________________________________________________
dropout (Dropout)                               (128, 128)                                      0
_____________________________________________________________________________________________________________
dense   (Dense)                                 (128, 10)                                       1290
=============================================================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
```

Compare models (with/without dropout)



Models' performances

- Evaluation both models in the test set (10000 digits)
- Without dropout:

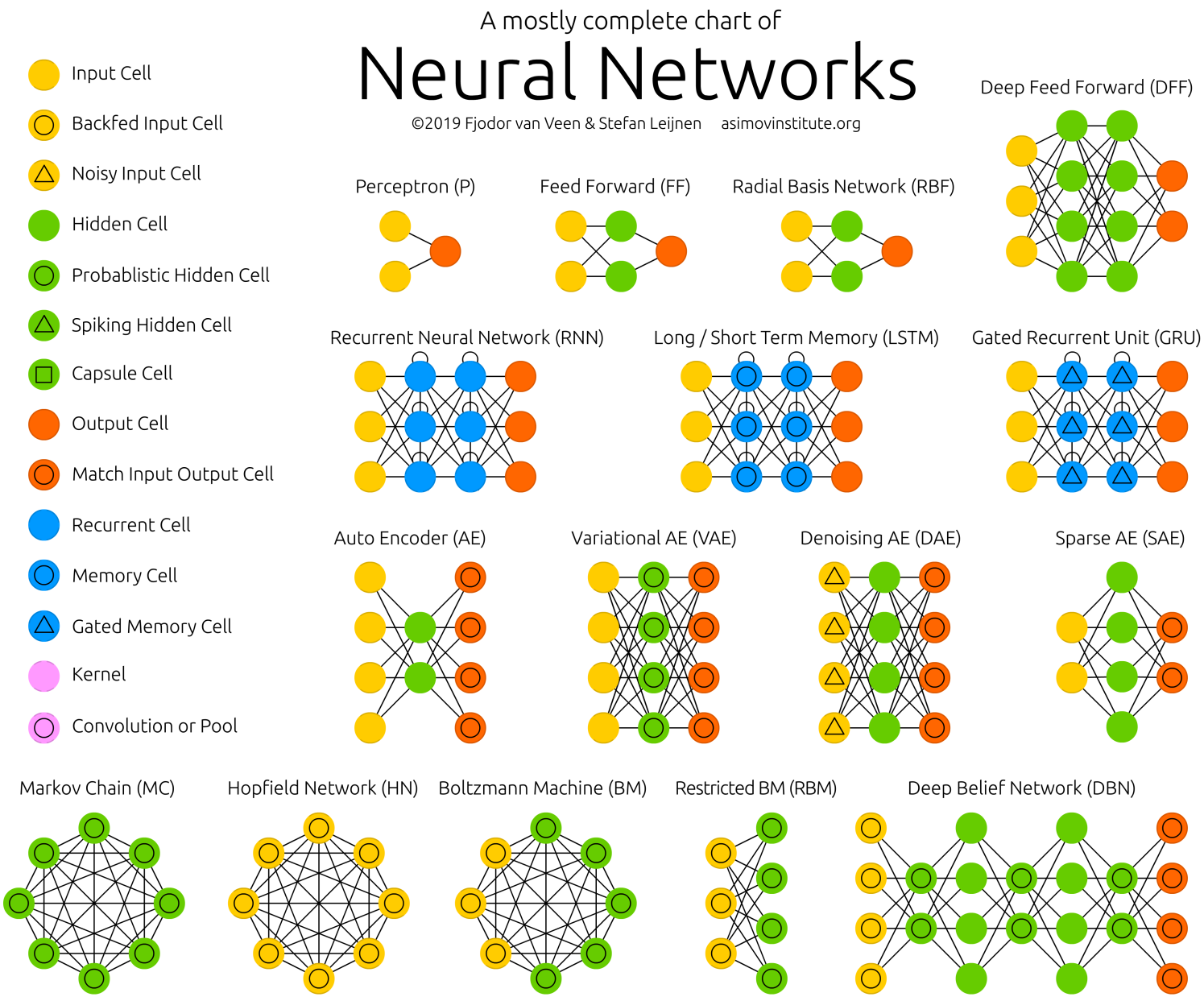
$$\text{accuracy} = 0.9801000$$

- With dropout:

$$\text{accuracy} = 0.98119998$$

- Human error rates around 2–2.5% (accuracy 97.5-98%)

Next lectures ...



Next lectures ...