Deep Learning Methods for NLP

Machine Learning for Natural Language Processing, ENSAE 2022

Lecture 3

Benjamin Muller, INRIA Paris

Lectures Outline

1. The Basics of Natural Language Processing (February 1st)
2. Representing Text with Vectors (February 1st)
3. **Deep Learning Methods for NLP (February 8th)**
4. Language Modeling (February 8th)
5. Sequence Labelling (Sequence Classification) (February 15th)
6. Sequence Generation Tasks (February 15th)

Today Lecture Outline

- Deep Learning Framework
- The Multi-Layer Perceptron
- Recurrent Neural Network
- Attention Mechanism
- Self-Attention Mechanism and the Transformer Architecture

Motivations

So far, we have seen, **techniques to represent tokens with vectors**

Given a certain representations of tokens:
➔ **How can we model a sequence of tokens to perform a specific task?**

In the past 10 years, a “new” class of machine learning techniques has become very popular and successful in NLP: **Deep Learning**

In this session, we introduce Deep Learning with a focus on the methods used in NLP

Framework

We want to model $(X_1, .., X_T)$ i.e. find the correct label *Y*

$$\begin{aligned} dnn_\theta : \quad & \mathbb{R}^{d,T} \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p \\ & (X_1, .., X_T) \mapsto \hat{Y} \end{aligned}$$

- Output space is $\mathbb{R}^p$ for **Regression** tasks
- Output space is $[|0, K|]^p$ for **Classification** tasks

Framework

We want to model $(X_1, .., X_T)$ i.e. find the correct label *Y*

$$dnn_\theta : \quad \mathbb{R}^{d,T} \quad \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p$$
$$(X_1, .., X_T) \mapsto \hat{Y}$$

Questions: when we do Deep Learning…

- **How do we define dnn_θ ?**
- **How do we train dnn_θ with data ?**

Framework

Given a sequence of vectors $(X_1, .., X_T)$ we want to predict *Y*

$$dnn_\theta : \quad \mathbb{R}^{d,T} \quad \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p$$
$$(X_1, .., X_T) \mapsto \hat{Y}$$

Most Deep Learning Models (all the ones we will use in this course):

- are **parametric** (i.e. $\theta \in \mathbb{R}^D$)
- defined as a **composition of "simple" functions (linear & non-linear)**
- are trained in an **end-to-end** fashion with **backpropagation**

NB: In Deep Learning, **the parametrization of *dnn*** is called **the Architecture**

Different Types of Architecture

How can we define our predictive function dnn_θ ?

➔ Multi-Layer Perceptron

➔ Recurrent Layers

➔ Attention Layers

➔ Self-Attention Layers (in a Transformer Architecture)

Different Types of Architecture

How can we define our predictive function dnn_θ ?

- ➔ Multi-Layer Perceptron
- ➔ Recurrent Layers
- ➔ Attention Layers
- ➔ Self-Attention Layers (in a Transformer Architecture)

How do we **train our model**? (i.e. estimate the parameters of the model)

- ➔ **Stochastic Gradient Descent** also called **backpropagation** in this context

The MultiLayer Perceptron (MLP)

aka "the Most simple Deep Learning Architecture"

The **MLP** works **on unidimensional data** (e.g. dimension *d*)

We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$))

$$\begin{aligned} dnn_\theta : \quad & \mathbb{R}^d \to \mathbb{R}^2 \\ & X \mapsto \hat{Y} \end{aligned}$$

The MultiLayer Perceptron (MLP)

aka "the Most simple Deep Learning Architecture"

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$))

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta\times d}, b_1 \in \mathbb{R}^{\delta}, W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \to \mathbb{R}^\delta$

The MultiLayer Perceptron (MLP)

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta\times d}, b_1 \in \mathbb{R}^{\delta}, W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \to \mathbb{R}^\delta$

➔ This model is a ***2-layer* MLP** model

The MultiLayer Perceptron (MLP)

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta\times d}, b_1 \in \mathbb{R}^{\delta}, W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \to \mathbb{R}^\delta$

➔ This model is a ***2-layer* MLP** model
➔ With **1 *hidden layer*** of dimension δ

The MultiLayer Perceptron (MLP)

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta\times d}, b_1 \in \mathbb{R}^{\delta}, W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \rightarrow \mathbb{R}^{\delta}$

➔ This model is a ***2-layer* MLP** model
➔ With **1 *hidden layer*** of dimension δ
➔ Taking as input a vector of **dimension *d*** to output a vector of **dimension 2**

The MultiLayer Perceptron (MLP)

The **MLP** works **on unidimensional data** (e.g. dimension *d*)
We present the **MLP in the regression case** (e.g. output space is $\mathbb{R}^2$

$$\boxed{dnn_{(W_1,b1,W_2,b_2)}(X) = W_2\varphi_1(W_1X + b_1) + b_2}$$

W_1, b_1, W_2 and b_2 are trainable parameters. $W_1 \in \mathbb{R}^{\delta\times d}, b_1 \in \mathbb{R}^{\delta}, W_2 \in \mathbb{R}^{2\times\delta}$ and $b_2 \in \mathbb{R}$

φ_1 is a fixed non-linear function, $\varphi_1 : \mathbb{R}^d \rightarrow \mathbb{R}^{\delta}$

- ➔ This model is a ***2-layer* MLP** model
- ➔ With **1 *hidden layer*** of dimension δ
- ➔ Taking as input a vector of ***dimension d*** to output a vector of **dimension 2**
- ➔ Such a model is also referred to as a ***Feed-Forward* Neural Network (FNN)**

The MultiLayer Perceptron: Diagram View

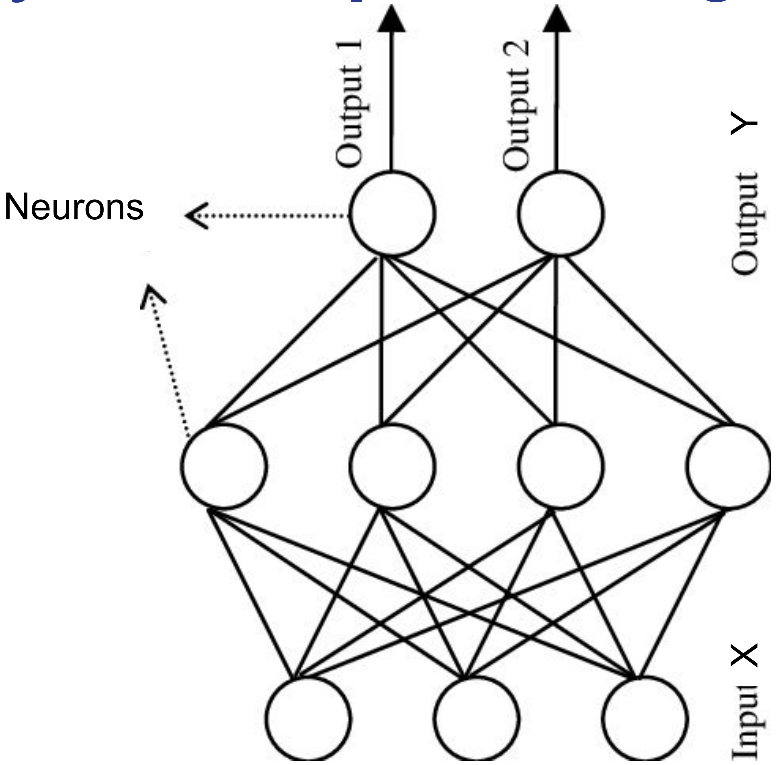
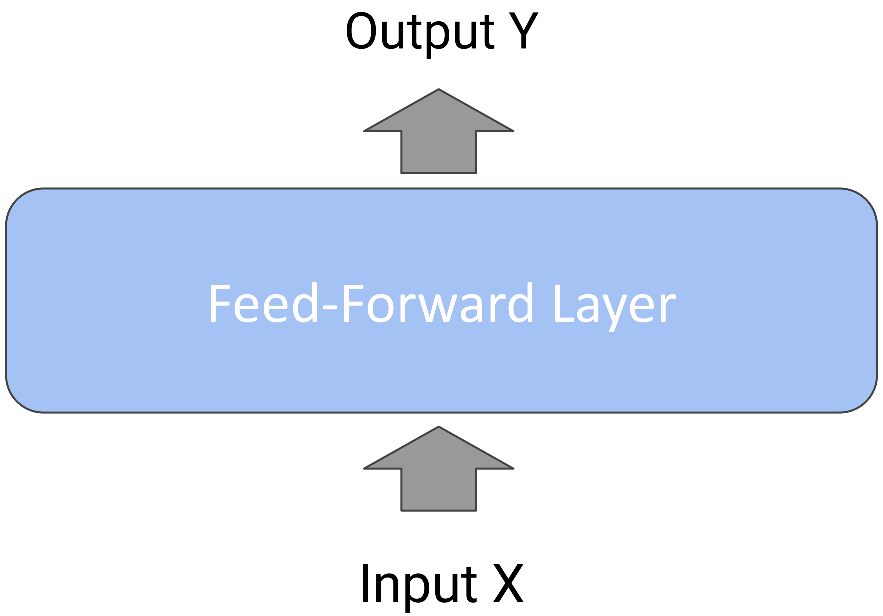


Figure from **(R. Rezvani et. al. 2012)**

In Deep Learning, it is usual to represent equations **with diagrams**

The MultiLayer Perceptron: Diagram View



In Deep Learning, it is usual to represent equations **with diagrams**

The MultiLayer Perceptron:

We have defined a 2-layers MLP model
We can define in the same way a **3-layers**, **4-layers**, **L-layers** MLP

$$\boxed{dnn_{(W_i\, b_i, i \in [|1,L|])}(X) = W_L \varphi_{L-1}(...\varphi_2 \circ W_2 \varphi_1(W_1 X + b_1) + b_2)...) + b_L}$$

W_l and b_l are trainable parameters. $W_l \in \mathbb{R}^{\delta_{l-1} \times \delta_l}$, $b_l \in \mathbb{R}^{\delta_l}$, with $\delta_l \in \mathbb{N}^*, \forall\, l \in [|1, L|]$

φ_l fixed non-linear functions, $\varphi_l : \mathbb{R}^{\delta_{l-1}} \rightarrow \mathbb{R}^{\delta_l}, \forall\, l \in [|1, L-1|]$

The MultiLayer Perceptron

The same equation with a loop…

$$\boxed{\begin{array}{l} h_{i+1} = \varphi_i(W_i h_i + b_i), \forall\, i \in [|1, L-1|] \\ \text{with } h_1 = X \text{ and } \hat{Y} = dnn(X) = h_L \end{array}}$$

W_l and b_l are trainable parameters. $W_l \in \mathbb{R}^{\delta_{l-1} \times \delta_l}$, $b_l \in \mathbb{R}^{\delta_l}$, with $\delta_l \in \mathbb{N}^*, \forall\, l \in [|1, L|]$

φ_l fixed non-linear functions, $\varphi_l : \mathbb{R}^{\delta_{l-1}} \rightarrow \mathbb{R}^{\delta_l}, \forall\, l \in [|1, L-1|]$

The MultiLayer Perceptron

The same equation with a loop…

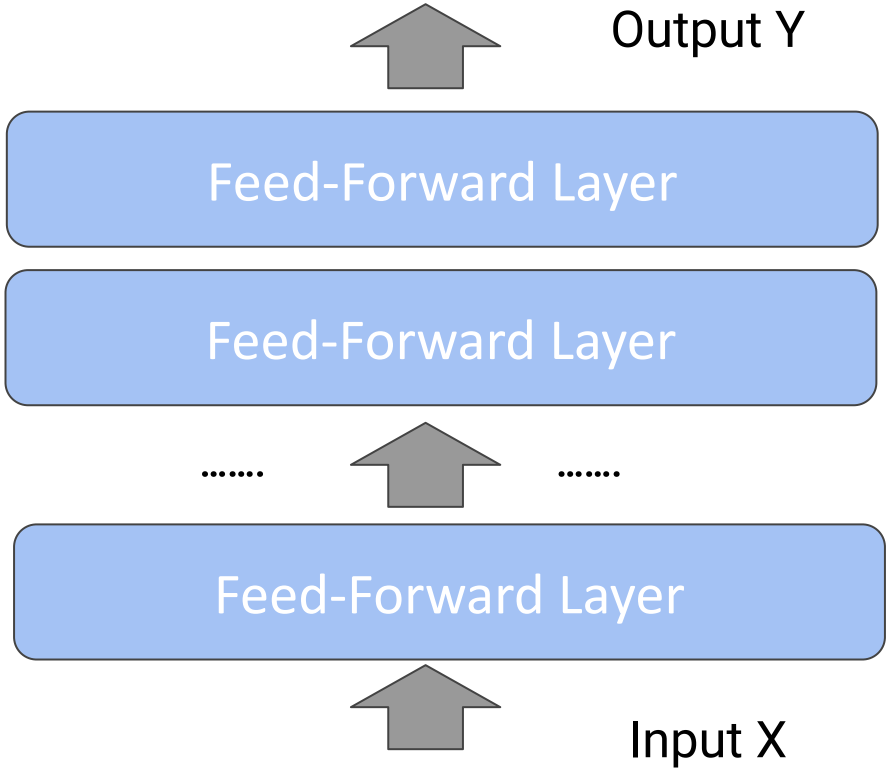
$$\boxed{\begin{aligned} & h_{i+1} = \varphi_i(W_i h_i + b_i), \forall\, i \in [|1, L-1|] \\ & \text{with } h_1 = X \text{ and } \hat{Y} = dnn(X) = h_L \end{aligned}}$$

W_l and b_l are trainable parameters. $W_l \in \mathbb{R}^{\delta_{l-1} \times \delta_l}$, $b_l \in \mathbb{R}^{\delta_l}$, with $\delta_l \in \mathbb{N}^*, \forall\, l \in [|1, L|]$

φ_l fixed non-linear functions, $\varphi_l : \mathbb{R}^{\delta_{l-1}} \rightarrow \mathbb{R}^{\delta_l}, \forall\, l \in [|1, L-1|]$

h_i are called hidden states ($h_i \in \mathbb{R}^{\delta_i}$).

The MultiLayer Perceptron: Diagram View



Output Activation Function for Classification

When we do a classification task the goal is to learn a distribution of probability on the output label space

To do so, **we usually use the softmax function** as the last activation function

$$softmax(s) = \left(\frac{e^{s_i}}{\sum_k e^{s_k}}\right)_{i \in [|1,K|]}, \text{ for } s \in \mathbb{R}^K$$

Loss Functions

Based on the task we aim at modeling, we can use:

For Regression: Mean-Square Error

$$l(y, \hat{y}) = \|y - \hat{y}\|_2^2 = \sum_i (y_i - \hat{y}_i)^2 \text{ assuming } y_i,\ \hat{y}_i \in \mathbb{R}$$

For Classification: Cross-Entropy Loss

$$l(y, \hat{y}) = CE(y, \hat{y}) = \sum_i y_i\, log(\hat{y}_i) \text{ assuming } y_i,\ \hat{y}_i \in\ [0, 1]$$

Most NLP tasks will be based on the **Cross-Entropy loss**

The MultiLayer Perceptron: Hyperparameters

- **Number of hidden layers**
- **Hidden layers dimensions**
- Initialization of the trainable parameters/weights

The MultiLayer Perceptron: Hyperparameters

- Number of hidden layers
- Hidden layers dimensions
- Initialization of the **trainable parameters/weights**

The MultiLayer Perceptron: Hyperparameters

- Number of hidden layers
- Hidden layers dimensions
- Initialization
- **Activation Functions**

The MultiLayer Perceptron: Hyperparameters

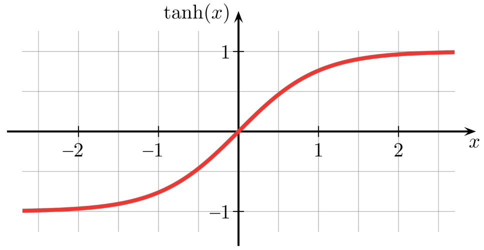
- Number of hidden layers
- Hidden layers dimensions
- Initialization
- **Activation Functions**

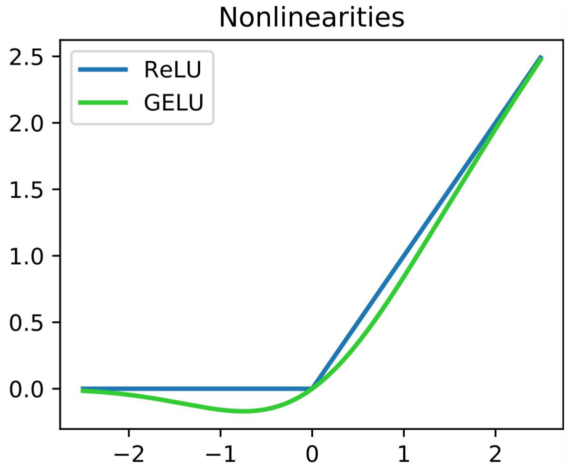
➢ They should be **non-linear**
➢ **Differentiable**
➢ **Standard ones are:**
Relu, tanh, sigmoid

The MultiLayer Perceptron: Hyperparameters

- Number of hidden layers
- Hidden layers dimensions
- Initialization
- **Activation Functions**

➢ They should be **non-linear**
➢ **Differentiable**
➢ **Standard ones are:**
Relu, tanh, sigmoid





The MultiLayer Perceptron: Hyperparameters

- Number of hidden layers
- Hidden layers dimensions
- Initialization
- Activation Functions

How to define them?

➔ Look for **best practices** to choose which are the best

➔ In most DL libraries, the **"good" hyperparameters are usually the default**

➔ If no best practices/default: **you have to find the best ones empirically**

Intuition

playground

Training Deep Learning Models

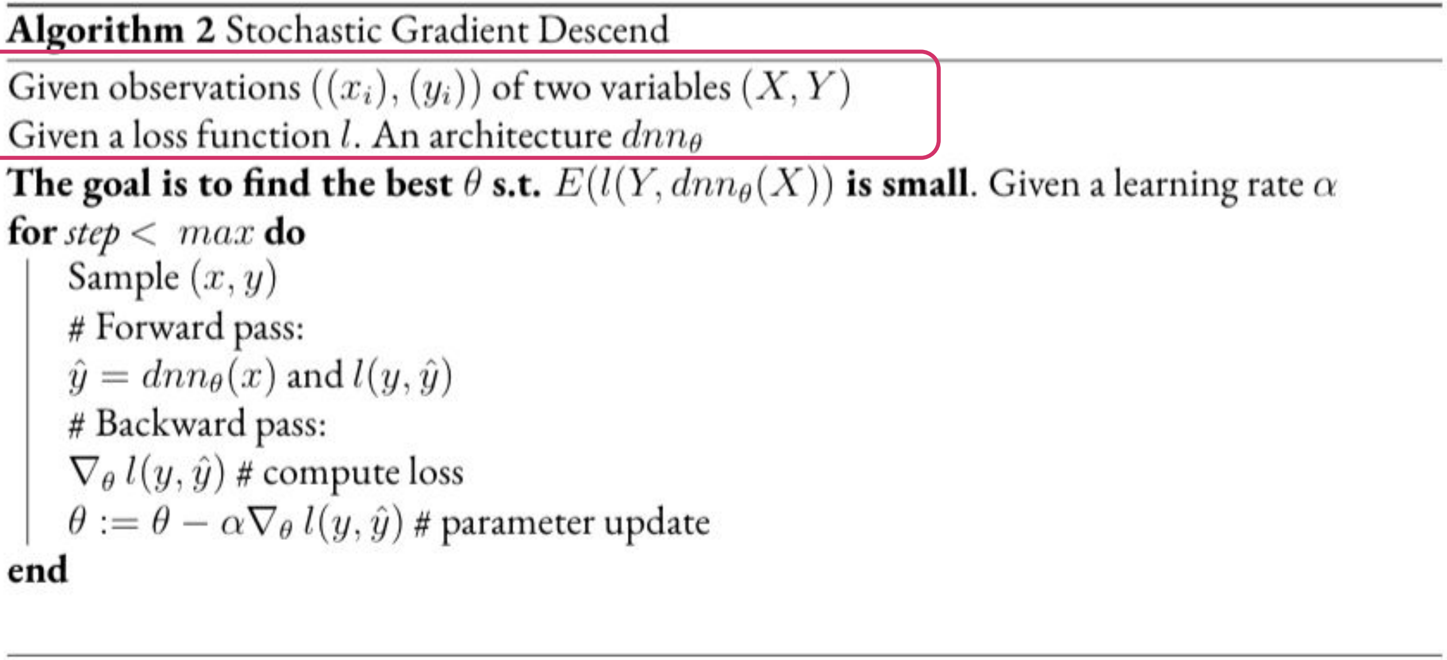
- Nearly all Deep Learning models are trained with (some version of) **Stochastic Gradient Descent (SGD)**

Stochastic Gradient Descent

- The goal is find the set of **parameters/weights** that **minimizes the loss function**
- To do so, SGD estimates the true gradient of a function with **one sample at time**
- **Repeat** this process multiple times

NB: in deep learning, we usually train all the parameters together **“end-to-end”**

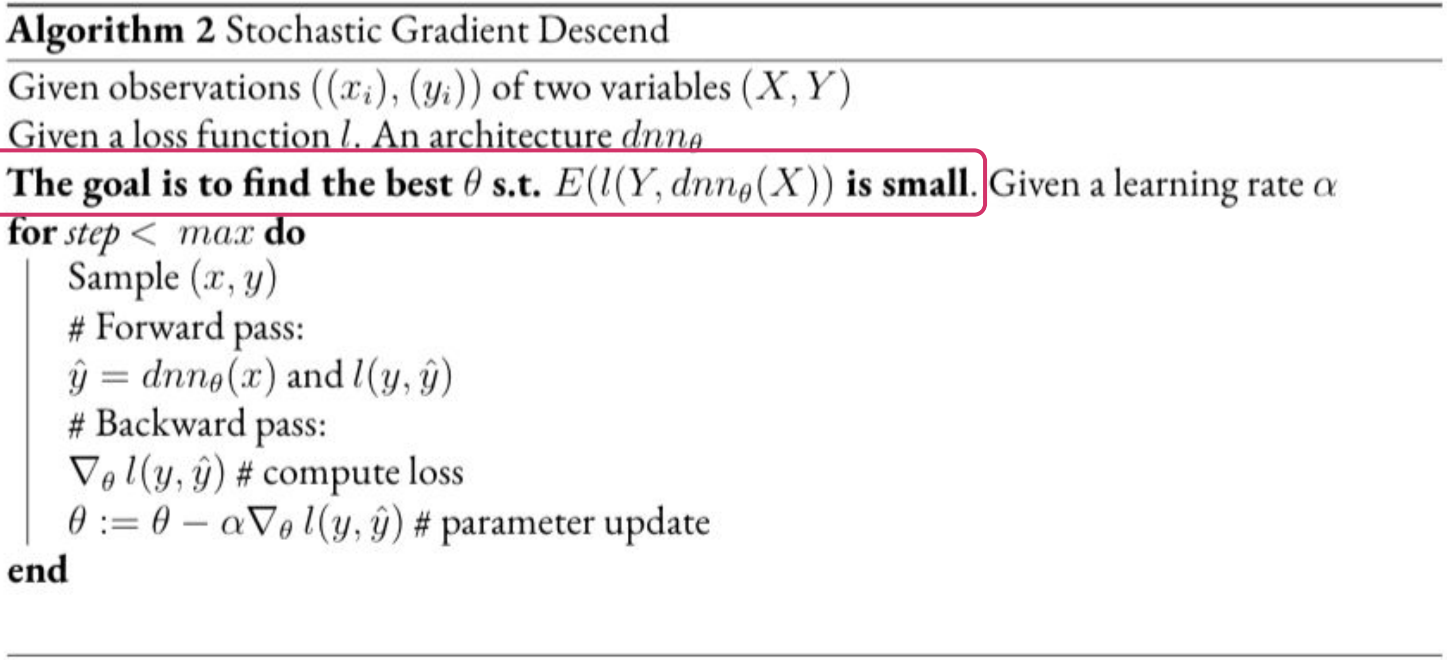
Stochastic Gradient Descent



Algorithm 2 Stochastic Gradient Descend

```
Given observations $((x_i), (y_i))$ of two variables $(X, Y)$
Given a loss function $l$. An architecture $dnn_\theta$
**The goal is to find the best** $\theta$ **s.t.** $E(l(Y, dnn_\theta(X))$ **is small**. Given a learning rate $\alpha$
**for** *step* $<$ $max$ **do**
    Sample $(x, y)$
    # Forward pass:
    $\hat{y} = dnn_\theta(x)$ and $l(y, \hat{y})$
    # Backward pass:
    $\nabla_\theta\, l(y, \hat{y})$ # compute loss
    $\theta := \theta - \alpha \nabla_\theta\, l(y, \hat{y})$ # parameter update
**end**
```

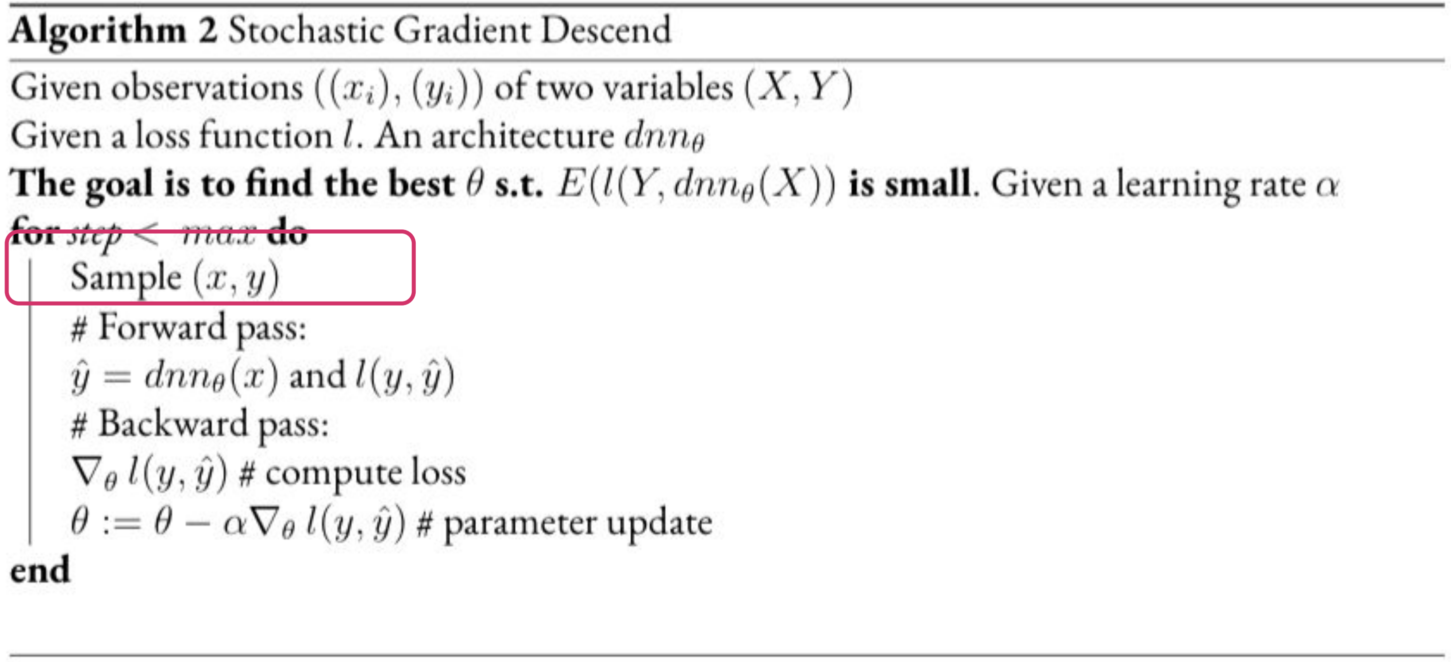
Stochastic Gradient Descent



Algorithm 2 Stochastic Gradient Descend

```
Given observations $((x_i), (y_i))$ of two variables $(X, Y)$
Given a loss function $l$. An architecture $dnn_\theta$
**The goal is to find the best $\theta$ s.t. $E(l(Y, dnn_\theta(X))$ is small.** Given a learning rate $\alpha$
**for** *step* $<$ $max$ **do**
    Sample $(x, y)$
    # Forward pass:
    $\hat{y} = dnn_\theta(x)$ and $l(y, \hat{y})$
    # Backward pass:
    $\nabla_\theta \, l(y, \hat{y})$ # compute loss
    $\theta := \theta - \alpha \nabla_\theta \, l(y, \hat{y})$ # parameter update
**end**
```

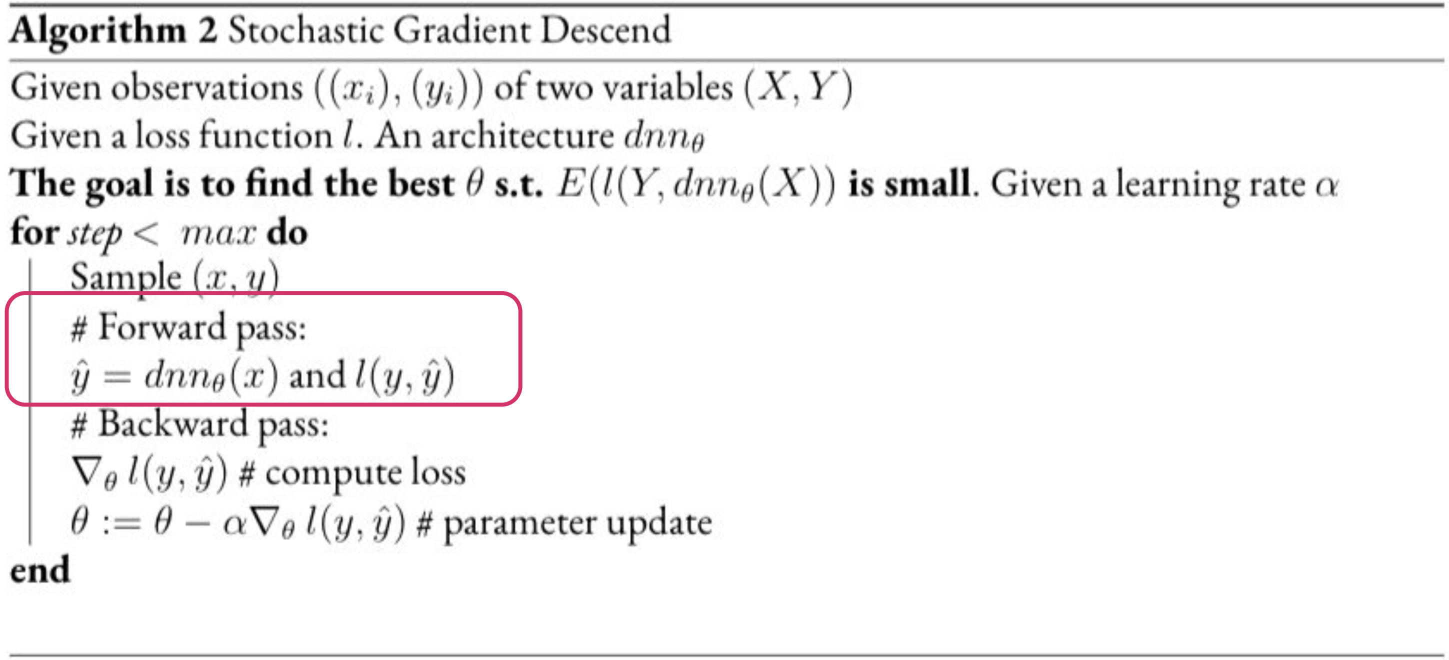
Stochastic Gradient Descent



Algorithm 2 Stochastic Gradient Descend

```
Given observations $((x_i), (y_i))$ of two variables $(X, Y)$
Given a loss function $l$. An architecture $dnn_\theta$
**The goal is to find the best** $\theta$ **s.t.** $E(l(Y, dnn_\theta(X))$ **is small**. Given a learning rate $\alpha$
**for** $step < max$ **do**
    Sample $(x, y)$
    # Forward pass:
    $\hat{y} = dnn_\theta(x)$ and $l(y, \hat{y})$
    # Backward pass:
    $\nabla_\theta \, l(y, \hat{y})$ # compute loss
    $\theta := \theta - \alpha \nabla_\theta \, l(y, \hat{y})$ # parameter update
**end**
```

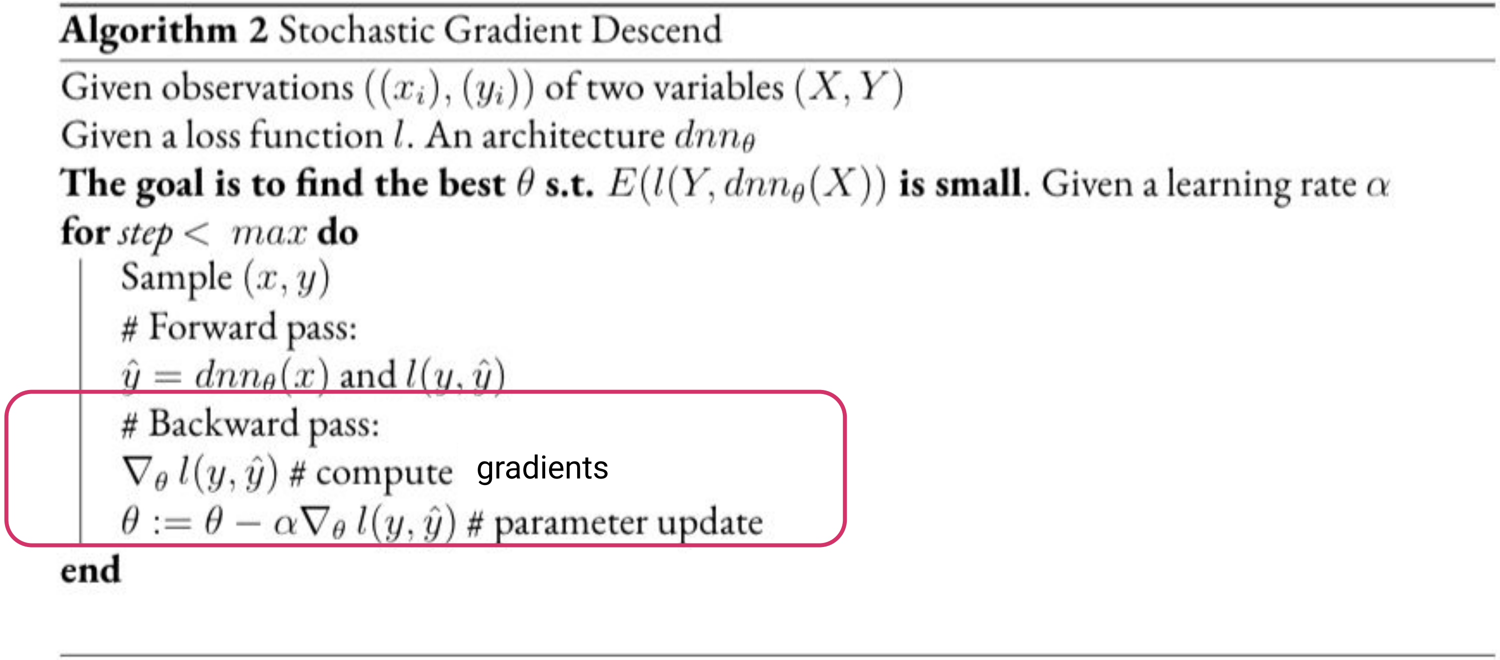
Stochastic Gradient Descent



Algorithm 2 Stochastic Gradient Descend

```
Given observations $((x_i), (y_i))$ of two variables $(X, Y)$
Given a loss function $l$. An architecture $dnn_\theta$
**The goal is to find the best** $\theta$ **s.t.** $E(l(Y, dnn_\theta(X)))$ **is small**. Given a learning rate $\alpha$
**for** *step* $< max$ **do**
    Sample $(x, y)$
    # Forward pass:
    $\hat{y} = dnn_\theta(x)$ and $l(y, \hat{y})$
    # Backward pass:
    $\nabla_\theta\, l(y, \hat{y})$ # compute loss
    $\theta := \theta - \alpha \nabla_\theta\, l(y, \hat{y})$ # parameter update
**end**
```

Stochastic Gradient Descent



Algorithm 2 Stochastic Gradient Descend

```
Given observations $((x_i), (y_i))$ of two variables $(X, Y)$
Given a loss function $l$. An architecture $dnn_\theta$
**The goal is to find the best** $\theta$ **s.t.** $E(l(Y, dnn_\theta(X)))$ **is small**. Given a learning rate $\alpha$
**for** *step* $<$ $max$ **do**
    Sample $(x, y)$
    # Forward pass:
    $\hat{y} = dnn_\theta(x)$ and $l(y, \hat{y})$
    # Backward pass:
    $\nabla_\theta\, l(y, \hat{y})$ # compute gradients
    $\theta := \theta - \alpha \nabla_\theta\, l(y, \hat{y})$ # parameter update
**end**
```

Stochastic Gradient Descent

Optimization Hyperparameters

Learning Rate

- Can be refined with **variable learning rate**
  *E.g. increasing during the first steps (**warmup**) then decreasing*

Number of steps

- Usually defined with based on the validation loss
  *When it stops decreasing we can stop training (=**early stopping**)*

Stochastic Gradient Descent

Optimizing large Deep Learning Models **is challenging**

- **Unstable training**
- **Overfitting**
- **Take a lot of steps/epochs**

To make training better, many refinement of the SGD have been proposed

- In practice, we (nowadays) **use the ADAM optimizer** (cf. Kingma et. al 2015)

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

We define a 1-hidden-layer MLP with a RELU activation function of dimension δ.

$$\boxed{\hat{y} = dnn_{W_1,W_2}(x) = W_2 \ \max(W_1 x, 0) \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}}$$

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

We define a 1-hidden-layer MLP with a RELU activation function of dimension δ.

$$\boxed{\hat{y} = dnn_{W_1,W_2}(x) = W_2 \ \max(W_1 x, 0) \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}}$$

➔ **Goal: Apply SGD to *dnn***

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

We define a 1-hidden-layer MLP with a ReLU activation function of dimension δ.

$$\hat{y} = dnn_{W_1,W_2}(x) = W_2 \ \max(W_1 x, 0) \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}$$

1. **Forward pass: Compute** $\hat{y}$

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

We define a 1-hidden-layer MLP with a ReLU activation function of dimension δ.

$$\hat{y} = dnn_{W_1,W_2}(x) = W_2 \max(W_1 x, 0) \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}$$

1. **Forward pass**
2. **Compute Gradients**

$$\nabla_{W_1} l(y, \hat{y}) \qquad \nabla_{W_2} l(y, \hat{y})$$

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

We define a 1-hidden-layer MLP with a ReLU activation function of dimension δ.

$$\boxed{\hat{y} = dnn_{W_1,W_2}(x) = W_2 \max(W_1 x, 0) \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}}$$

1. **Forward pass**
2. **Compute Gradients**
3. **Backward pass (parameter update)**

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

$$\boxed{\hat{y} = dnn_{W_1,W_2}(x) = W_2 \ \max(W_1 x, 0) \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}}$$

Idea: we use **the chain rule** to decompose **the gradient** starting **from the top layers**

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

$$\hat{y} = dnn_{W_1,W_2}(x) = W_2 \underbrace{\max(W_1 x, 0)}_{h_1} \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}$$

Compute Gradient

$$\nabla_{W_2} l(y, \hat{y}) = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2}$$

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

$$\hat{y} = dnn_{W_1,W_2}(x) = W_2 \underbrace{\max(W_1 x, 0)}_{h_1} \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}$$

Compute Gradient

$$\nabla_{W_2} l(y, \hat{y}) = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2} = 2(y - \hat{y})\, h_1$$

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

$$\boxed{\hat{y} = dnn_{W_1,W_2}(x) = W_2 \underbrace{\max(W_1 x, 0)}_{h_1} \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}}$$

Compute Gradient:

$$\nabla_{W_2} l(y, \hat{y}) = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2} = 2(y - \hat{y})\, h_1$$

$$\nabla_{W_1} l(y, \hat{y}) = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial W_2}$$

Stochastic Gradient Descent for MLP

Let $(X, Y) \in \mathbb{R}^d \times \mathbb{R}$, the MSE loss $l(y, \hat{y}) = (y - \hat{y})^2$.

$$\hat{y} = dnn_{W_1,W_2}(x) = W_2 \underbrace{\max(W_1 x, 0)}_{h_1} \text{ and } W_1 \in \mathbb{R}^{d\times\delta} \text{ and } W_2 \in \mathbb{R}^{1\times\delta}$$

Compute Gradient:

$$\nabla_{W_2} l(y, \hat{y}) = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_2} = 2(y - \hat{y})\, h_1$$

$$\nabla_{W_1} l(y, \hat{y}) = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial W_2} = 2(y - \hat{y})\, W_2 1_{W_1 X > 0}$$

Backpropagation and Deep Learning in practice

In practice, we use Deep Learning Libraries

- Define **the Architecture with *tensor* operators**
- Backpropagation is done **seamlessly using automatic differentiation**

Deep Learning & Backpropagation in practice

In practice, we use Deep Learning Libraries (e.g. pytorch, tensorflow, jax)

- Define **the Architecture with *tensor* operators**
- Backpropagation is done **seamlessly using automatic differentiation**

- Standard layers **are pre-implemented** (Feed-Forward Layers, LSTM, Attention, Self-Attention…)

See code example with pytorch

Recurrent Neural Network

Vanilla Recurrent Neural Network

We would like to model sequences (e.g. words) $(X_1, .., X_T) \text{ in } \mathbb{R}^{d,T}$

We can introduce **a recurrence relation** into our MLP to model it:

$$h_{i+1,t+1} = \varphi_i(W_i h_{i,t} + U_i h_{i+1,t} + +b_i), \forall\, i \in [|1, L-1|]$$
$$\text{with } h_{1,t} = X_t \text{ and } \hat{Y}_t = dnn(X_t) = h_{L,t} \forall\, t \in [|1, T-1|]$$

Recurrent Neural Network

Illustration of a 1-layer Recurrent Neural Network

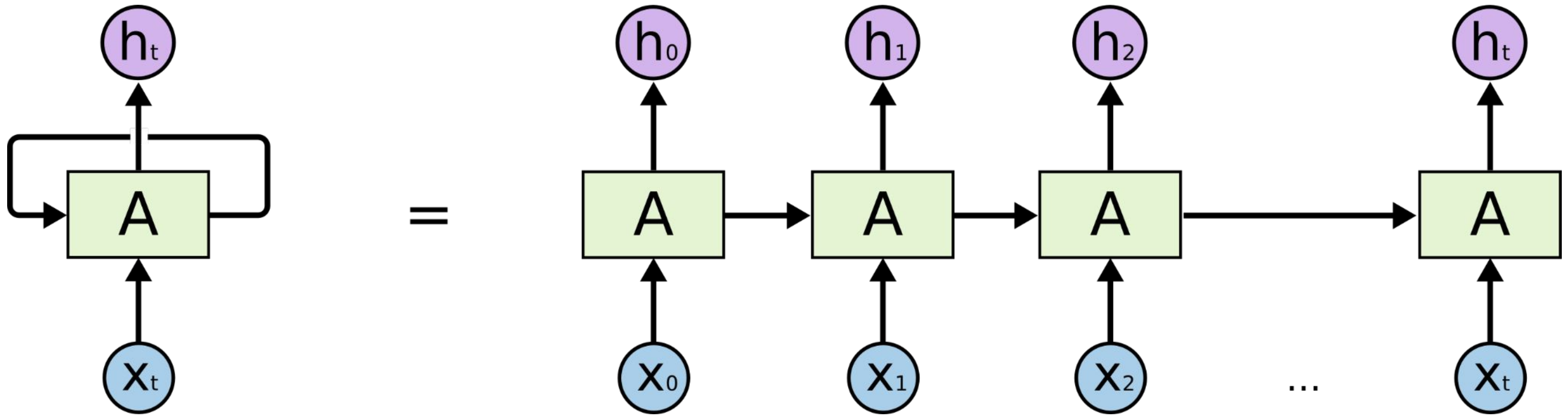


Figure from colah

Recurrent Neural Network

Illustration of a 1-layer Recurrent Neural Network

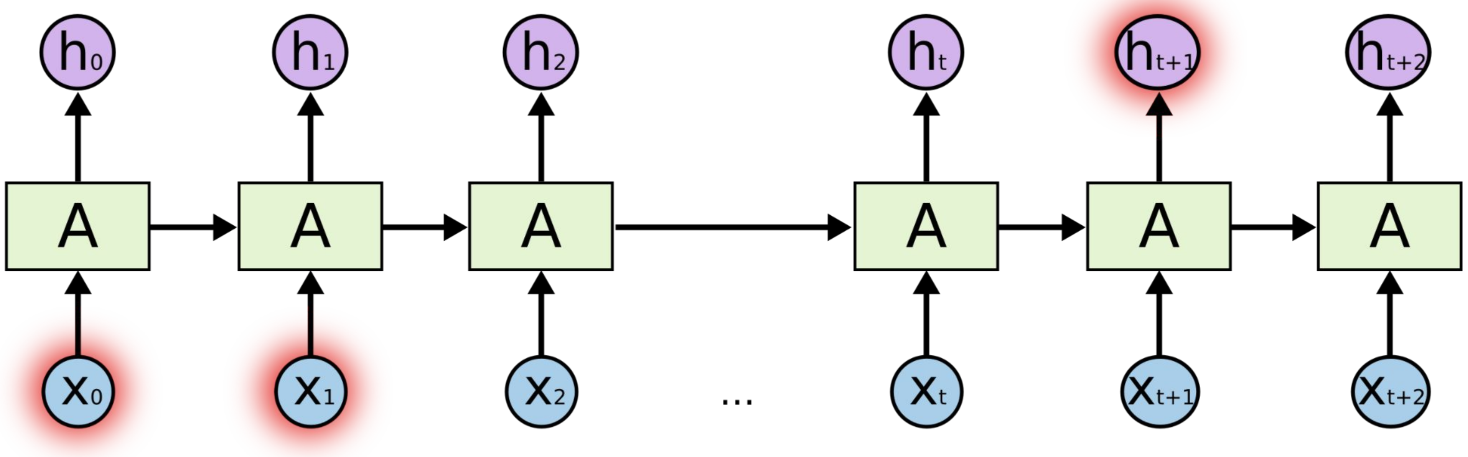


Figure from colah

Training Recurrent Neural Network

Recurrent Neural Network are trained with an extension of the Backpropagation algorithm

➔ Backpropagation Through Time (BPTT)

BPTT follows exactly the same ideas as backpropagation

- SGD
- Chain Rule starting from the last layer and the last hidden state
- **With extra derivative dependencies between state *t and t+1***

Limits of Recurrent Neural Networks

Vanilla Recurrent Neural Network have trouble to capture long-term dependencies

Idea:

- Encode **explicitly in a vector a "memory" in the recurrent architecture**
- Control what is memorized and forgotten
- Train all those parameters **end-to-end**

LSTM: Long-Short Term Memory Network

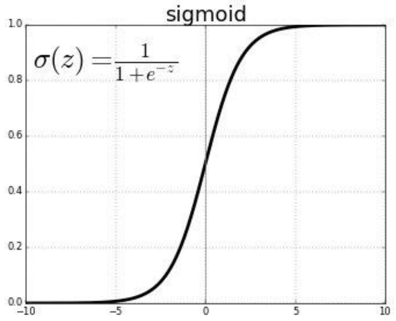
Introduce a memory vector C_t

C_t is designed to **capture long term dependencies**

The output state h_t ***of each LSTM cell is based on*** C_t ***and an output gate*** o_t

$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$



LSTM: Long-Short Term Memory Network

Introduce a memory vector C_t

C_t is designed to **capture long term dependencies**

C_t is define recurrently based on the previous step and the input and the forget gate. Those gates control what is memorized and forgotten.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

LSTM: Long-Short Term Memory Network

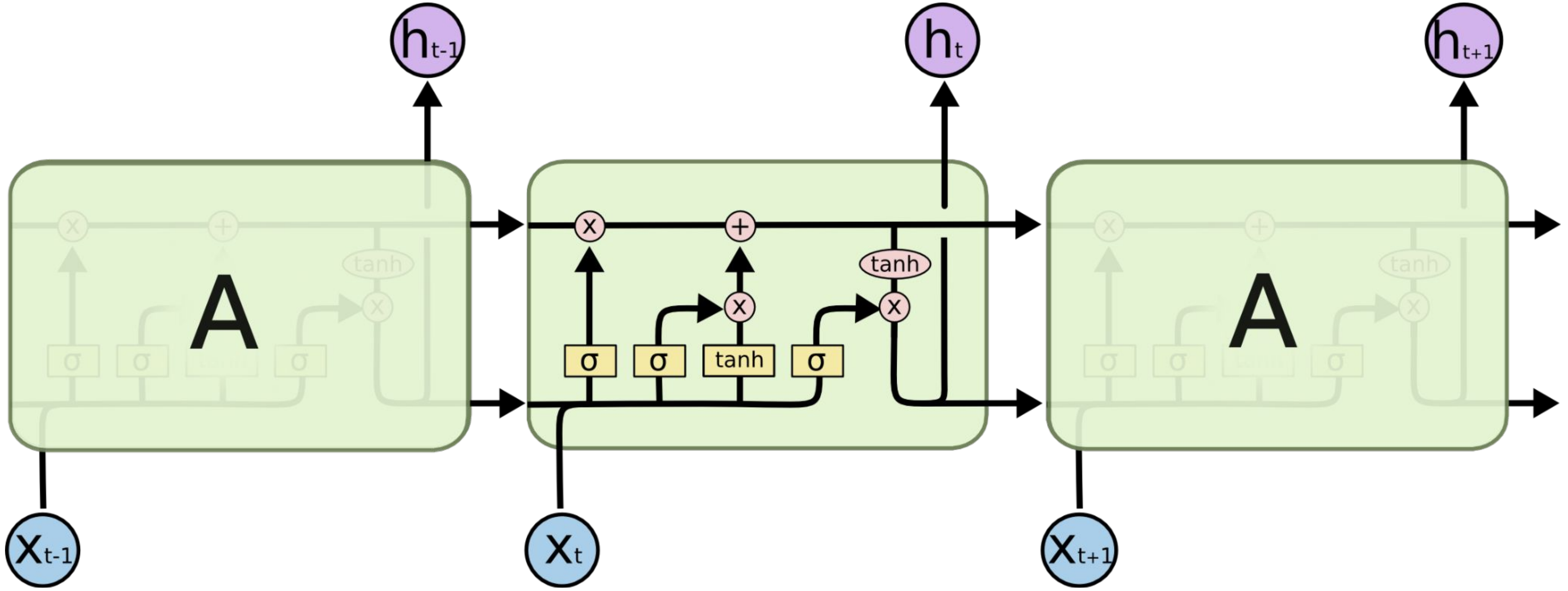


Figure from colah

LSTM: Long-Short Term Memory Network

- We train LSTM with Backpropagation (through time)
- LSTM cells are usually combined with Feed-Forward Layers

NB: Until recently (2018), LSTM-based models were delivering **State-of-the-art performance for most sequence modelling tasks**

Attention Mechanism

Motivation for Attention Mechanisms

- The Deep Learning Architecture that we have seen so far are **hard to interpret** **(black-box)**
- Recurrent Network provide a fixed vector encoding of a sequence at each step

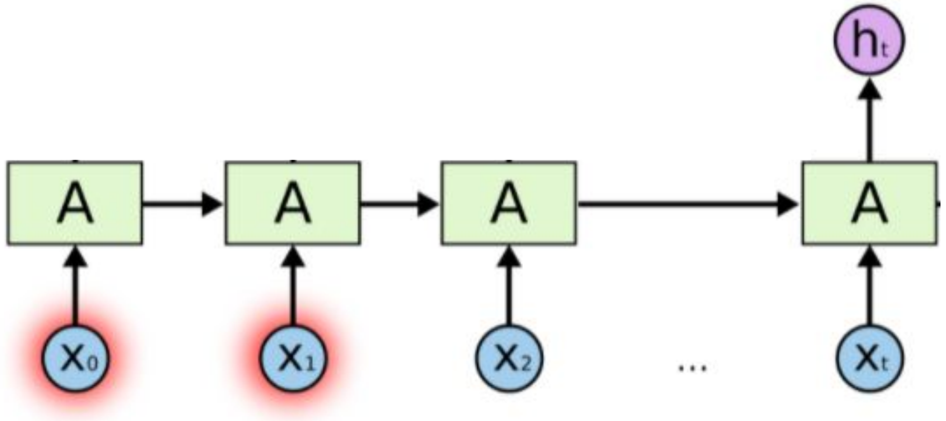
➔ **Attention Mechanisms**

Attention Mechanism for Sequence Classification

We want to classify (X0, Xt) sequences (e.g. sentiment analysis)

Solution 1: Use a LSTM model → Problem (not interpretable)

Feed to Softmax Layer for Classification

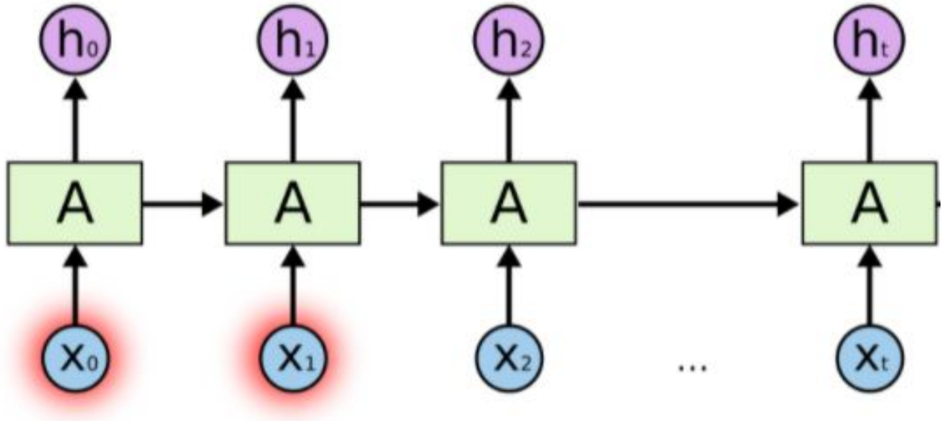


Attention Mechanism for Sequence Classification

We want to classify (X0, Xt) sequences (e.g. sentiment)

Solution 2: Integrate an Attention Mechanism to interpret what input impacts the prediction

➔ **Learn a ponderation/weighting of the hidden states ht**



Attention Mechanism for Sequence Classification

We want to classify (X0, Xt) sequences (e.g. sentiment)

How to learn this weighting?

1. Define a specific type of layer to learn the ponderation
2. Train this layer end-to-end with all the other parameters of the model

Attention Mechanism for Sequence Classification

We want to classify (X0, Xt) sequences (e.g. sentiment)

How to learn this weighting?

Given $(h_1, .., h_T)$ hidden representations of $(x_1, .., x_T)$ (e.g. output of a LSTM Layer).

$$\boxed{q_i = tanh(W_a h_i + b_a), \text{ with } W_a \in \mathbb{R}^{\delta \times \delta_a}}$$

$$s_t = \frac{e^{q_t q_T}}{\sum_j e^{q_j q_T}}, \text{ i.e. } \sum_{t \in [|1,T|]} s_i = 1$$

$$\tilde{h_T} = \sum_{t \in [|1,T|]} s_t . h_t$$

Attention Mechanism for Sequence Classification

We want to classify (X0, Xt) sequences (e.g. sentiment)

How to learn this weighting?

Given $(h_1, .., h_T)$ hidden representations of $(x_1, .., x_T)$ (e.g. output of a LSTM Layer).

$$q_i = tanh(W_a h_i + b_a), \text{ with } W_a \in \mathbb{R}^{\delta \times \delta_a}$$

$$\boxed{s_t = \frac{e^{q_t q_T}}{\sum_j e^{q_j q_T}}, \text{ i.e. } \sum_{t \in [|1,T|]} s_i = 1}$$

$$\tilde{h_T} = \sum_{t \in [|1,T|]} s_t . h_t$$

Attention Mechanism for Sequence Classification

We want to classify (X0, Xt) sequences (e.g. sentiment)

How to learn this weighting?

Given $(h_1, .., h_T)$ hidden representations of $(x_1, .., x_T)$ (e.g. output of a LSTM Layer).

$$q_i = tanh(W_a h_i + b_a), \text{ with } W_a \in \mathbb{R}^{\delta \times \delta_a}$$

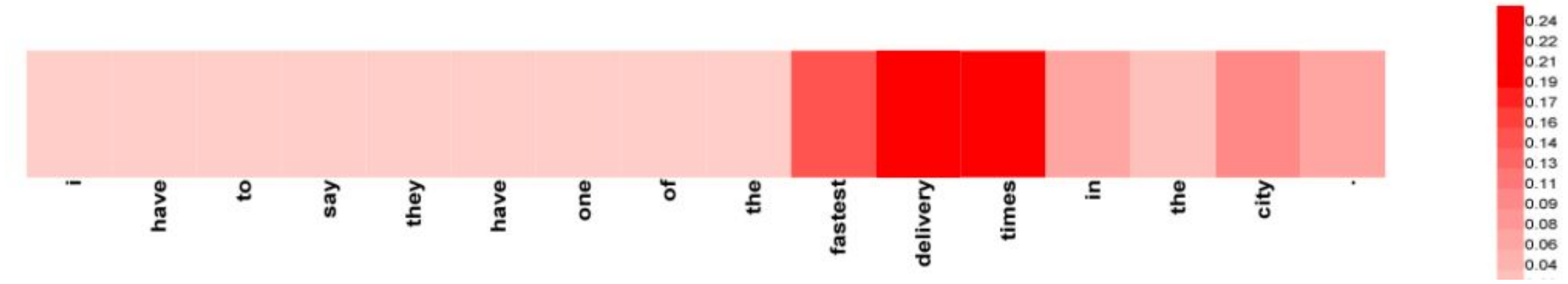
$$s_t = \frac{e^{q_t q_T}}{\sum_j e^{q_j q_T}}, \text{ i.e. } \sum_{t \in [|1,T|]} s_i = 1$$

$$\boxed{\tilde{h_T} = \sum_{t \in [|1,T|]} s_t . h_t}$$

Attention Mechanism for Sequence Classification

We want to classify (X0, Xt) sequences (e.g. sentiment classification)

After we trained the model, **Attention scores** can be used **to interpret the model** behavior and **what input vector impacted the decision**



(Wang et. al 2016)

Attention Mechanism for Sequence Classification

Many variant of Attention Mechanisms (in combination with LSTM layers) have been designed

Design Choices

- How to define the ***query vectors***?
- How to define the ***scoring function***?

Many variants exists but the principles are the same.

The Transformer Architecture

Attention might be all we need

Do we really need recurrent layers?

RNN models (such as vanilla RNN, LSTM…) were designed to model sequential data

Still, for most tasks, we **need both left and right context (e.g. sequence classification, sequence labelling..)**

Why not modelling sequences in a bi-directional way directly

➔ **Using Self-Attention Mechanism**

Self-Attention Layers

Given a sequence of input vectors $(x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $(h_{0,1}, .., h_{0,T})$).

Objective:

- Build a representation of the input vectors based on the **surrounding vectors** (both right-and left-context)

Idea:

- **No need of recurrent cells**

➔ **Self-Attention**

Self-Attention Layers: Intuition

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$)

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$).

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

- For a given vector h_t and its query vector q_t we want to build the new representation vector $\tilde{h}_t$
- Using the best ponderation of the information encoded in $(v_1, .., v_T)$
- This ponderation being computed by finding the key vectors in $(k_1, .., k_T)$ that are more similar to the query vector q_t (that encodes relevant information from h_t).

Self-Attention Layers: Intuition

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$.

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

- For a given vector h_t and its query vector q_t we want to build the new representation vector $\tilde{h}_t$
- Using the best ponderation of the information encoded in $(v_1, .., v_T)$
- This ponderation being computed by finding the key vectors in $(k_1, .., k_T)$ that are more similar to the query vector q_t (that encodes relevant information from h_t).

Self-Attention Layers: Intuition

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$.

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

- For a given vector h_t and its query vector q_t we want to build the new representation vector $\tilde{h}_t$
- Using the best ponderation of the information encoded in $(v_1, .., v_T)$
- This ponderation being computed by finding the key vectors in $(k_1, .., k_T)$ that are more similar to the query vector q_t (that encodes relevant information from h_t).

Self-Attention Layers: Intuition

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$).

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

- For a given vector h_t and its query vector q_t we want to build the new representation vector $\tilde{h}_t$
- Using the best ponderation of the information encoded in $(v_1, .., v_T)$
- This ponderation being computed by finding the key vectors in $(k_1, .., k_T)$ that are more similar to the query vector q_t (that encodes relevant information from h_t).

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$.

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

$$\boxed{q_t = W_Q h_t \ , \forall\, t \in [|1, T|] \text{ with } W_Q \in \mathrm{R}^{\delta_q \times \delta}}$$

$$k_t = W_K h_t \ , \forall\, t \in [|1, T|] \text{ with } W_K \in \mathrm{R}^{\delta_k \times \delta}$$

$$v_t = W_V h_t \ , \forall\, t \in [|1, T|] \text{ with } W_V \in \mathrm{R}^{\delta_v \times \delta}$$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$.

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

$$q_t = W_Q h_t \ , \forall\, t \in [|1, T|] \text{ with } W_Q \in \mathrm{R}^{\delta_q \times \delta}$$

$$\boxed{k_t = W_K h_t \ , \forall\, t \in [|1, T|] \text{ with } W_K \in \mathrm{R}^{\delta_k \times \delta}}$$

$$v_t = W_V h_t \ , \forall\, t \in [|1, T|] \text{ with } W_V \in \mathrm{R}^{\delta_v \times \delta}$$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$).

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

$$q_t = W_Q h_t \ , \forall\, t \in [|1, T|] \text{ with } W_Q \in \mathrm{R}^{\delta_q \times \delta}$$

$$k_t = W_K h_t \ , \forall\, t \in [|1, T|] \text{ with } W_K \in \mathrm{R}^{\delta_k \times \delta}$$

$$\boxed{v_t = W_V h_t \ , \forall\, t \in [|1, T|] \text{ with } W_V \in \mathrm{R}^{\delta_v \times \delta}}$$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$).

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

$$\tilde{H} = softmax(\frac{Q\,K^T}{\sqrt{\delta_K}})V$$

i.e. $\tilde{h_t} = softmax(\frac{q_t\,K^T}{\sqrt{\delta_K}}).V = \sum_{t'} s_{t'}\,v_{t'}$ with $s_{t'} = \frac{e^{q_{t'}k_t}}{\sum_t e^{q_{t'}k_t}}$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$).

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

$$\boxed{\tilde{H} = softmax(\frac{Q\,K^T}{\sqrt{\delta_K}})V}$$

i.e. $\tilde{h_t} = softmax(\frac{q_t\,K^T}{\sqrt{\delta_K}}).V = \sum_{t'} s_{t'}\,v_{t'}$ with $s_{t'} = \frac{e^{q_{t'}k_t}}{\sum_t e^{q_{t'}k_t}}$

Self-Attention Layers

Given a sequence of input vectors $X = (x_1, .., x_T) \in \mathbb{R}^\delta$ (noted $H = (h_{0,1}, .., h_{0,T})$).

We build 3 new vectorial representation of our sequence $H = (h_1, .., h_T)$.

The *query* $Q = (q_1, .., q_T)$, the *key* $K = (k_1, .., k_T)$ and the *value* $V = (v_1, .., v_T)$ vectors.

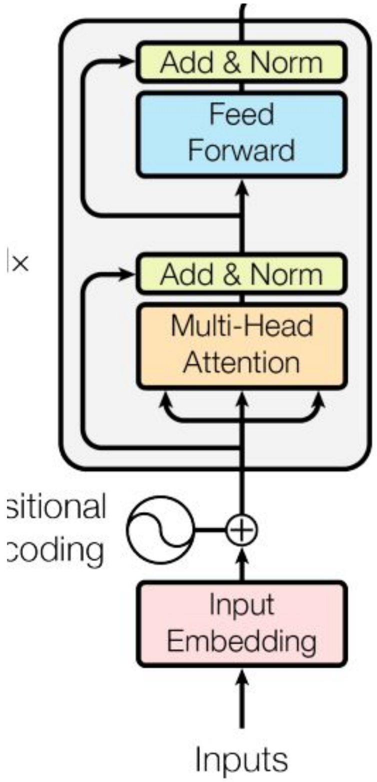
$$\tilde{H} = softmax(\frac{Q\,K^T}{\sqrt{\delta_K}})V$$

$$\text{i.e.} \quad \tilde{h_t} = softmax(\frac{q_t\,K^T}{\sqrt{\delta_K}}).V = \sum_{t'} s_{t'}\, v_{t'} \text{ with } \; s_{t'} = \frac{e^{q_{t'}k_t}}{\sum_t e^{q_{t'}k_t}}$$

The Transformer Architecture

The Transformer Architecture is

- Stack of [Self-Attention + FF Layer]
- With Skip-Layer and Normalization Layers in between
- Encoding the position with positional vector



Positional Embedding Vector

- **Limitation:** self attention does not take position into account!
- Indeed, shuffling the input gives the same results

- **Solution:** add position encodings.
- Replace the matrix $\mathbf{W}$ by $\mathbf{W} + \mathbf{E}$, where $\mathbf{E} \in \mathbb{R}^{d \times T}$

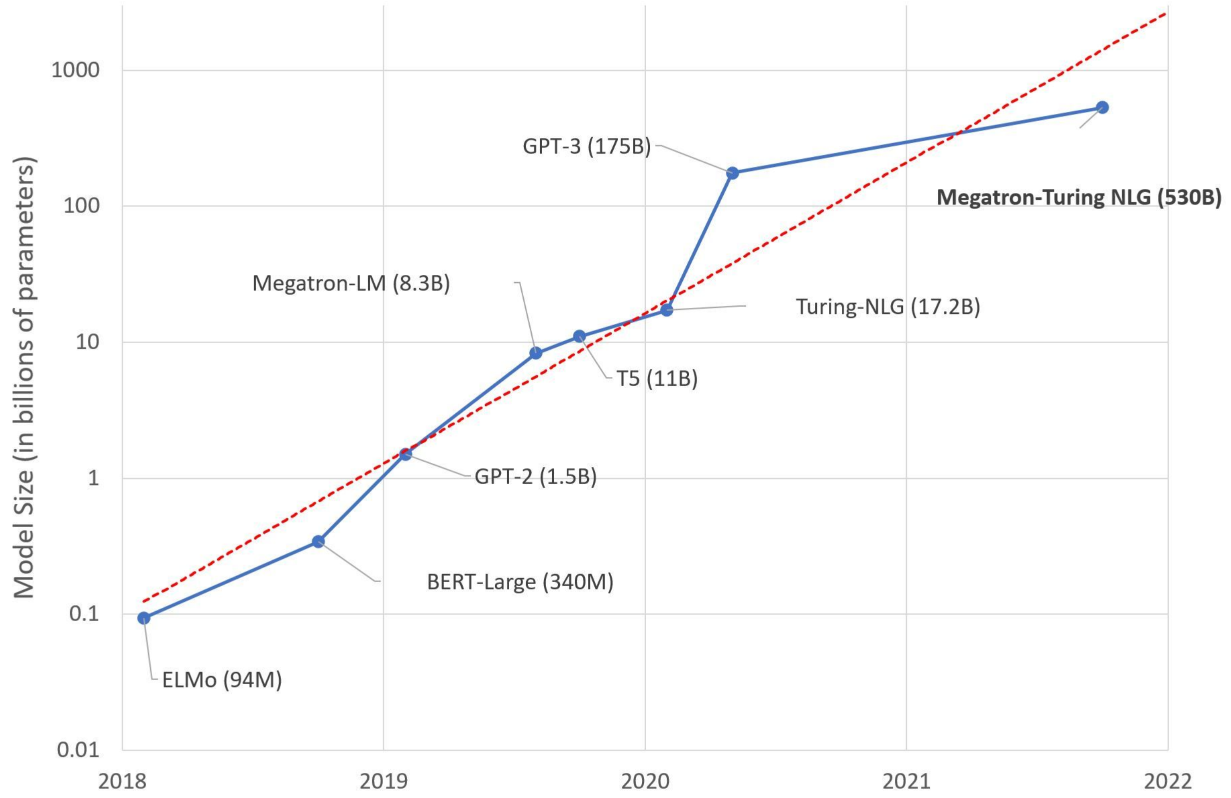
- $\mathbf{E}$ can be learned, or defined using sin and cos:

$$e_{2i,j} = \sin\left(\frac{j}{10000^{2i/d}}\right)$$
$$e_{2i+1,j} = \cos\left(\frac{j}{10000^{2i/d}}\right)$$

Scaling Laws Intuition

- The larger the dimension of the weight matrices
- The larger the number of parameters in the model
- The more "expressive" is the model
- The better it will generalizes

Typical Architecture Sizes



Lecture Summary

Deep Learning is a powerful and general modelling approach

- **Designing Architectures**, i.e. composition of linear transformation and non-linear transformation (possibly including recurrences)
- All those transformations **should be differentiable**
- All the parameters of the model **are trained with backpropagation**
- **Toward a specific task** s.t. regression or classification
- All the hyperparameters are chosen based on **best-practices** or empirical research