Programming of Neural Networks

This time: The framework and Fully Connected Layers

Outline and Organization

- The next 4 weeks can be considered as a „mini-course“ in the lecture which consists of:

  1. Implementing a small Deep Learning Framework in a programming language of your choice, having (at least)

     1. Fully Connected layers
     2. 2D Convolutional layers
     3. Pooling Layers
     4. Long Short Term Memory Layers

  2. Deriving the required equations for these types of layers

Outline and Organization

- After this course we expect you to:

1. Be able to perform the network operations (forward, backward, parameter update) by hand (yes, this contains „math“)

2. Derive the according equations of new network types yourself („Basicly you understand Backpropagation“)

Outline and Organization

- Outline:

  - **Week 1**: The general layout of the framework and fully connected layer
    - Exercise: Supervised Exercise, similar to the JPP

  - **Week 2**: 2D-Convolution De-mystified
    - Exercise: Supervised Exercise, similar to the JPP

  - **Week 3**: Recurrent Neural Networks using LSTM‘s
    - Exercise: Supervised Exercise, similar to the JPP

  - **Week 4**: Automatic Differentiation and Summary: ➔ Would be cool if someone of you could present his/her code to the other students, so we could all discuss and become better programmers!
    - Exercise: Typical exam questions related to this block

Disclaimer

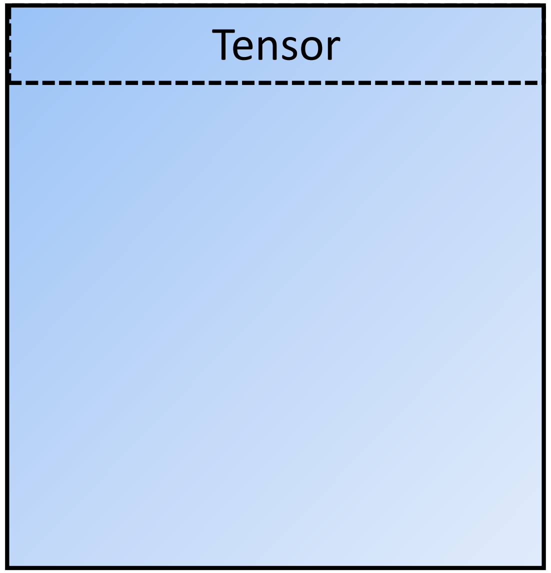
The following slides show an exemplary approach how to implement a „Deep Learning“ framework. The authors of this lecture do neither claim to present the best possible implementation guidelines nor to be expert programmers on their own.

No animals were harmed during the implementation of this framework

The Framework: Data structures

- The most crucial part of the implementation arises when asked of how to model the data in the framework:

➔ Introducing the „Tensor“ class



Data structures: Tensor

- In Deep Learning you will face data of various dimensions, example:
  - A vector (1D) $[0,1,2,3,4,5,6]$
  - A matrix (2D) $\begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix}$
  - An image with channels (3D) $\begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 4 & 6 \\ 5 & 7 \end{bmatrix} \begin{bmatrix} 8 & 10 \\ 9 & 11 \end{bmatrix}$
  - A convolution filter (4D)
  - Anything of the above with batchsize (5D)
  - …

Data structures: Tensor

- All of those have in common that they store „numbers“ of some sort, arranged in a given „Shape“
  - Storing all those values in a single array assures:
    - The values are aligned linearly in the RAM (fast access)
    - A single class can model data of any dimensionality
    - Can easily be reshaped (e.g. flattened)

Tensor
elements: FloatArray shape: Shape

Data structures: Shape

- Since shapes are very central to deep learning („what form your data is in“) they get their own class
  - Example, a Shape of [3,3] refers to a matrix with 3 rows and 3 columns

$$\begin{bmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}$$

  - A shape of [1,3] refers to a vector with 1 row and 3 columns

Y (columns, axis-1)

X(rows,axis-0)

$$[0 \quad 1 \quad 2]$$

~~Row-major~~ **Column–major** representation

see: https://en.wikipedia.org/wiki/Matrix_representation

Shape
axis: IntArray /volume: Int

The Network

- Now that we can handle N-dimensional data, we can think of the implementation of the network itself:

  - Brainstorming…

The Network - Inputlayer

- Hopefully, we found a consensus that a network consists of many „layers“:
  - Data is fed into the network using the input layer
  - Incoming objects can be of any type <T>, the network can only operate with Tensors

InputLayer<T>

Forward<T>(rawData:List<T>): List<Tensor>

The Network - Layers

- Hopefully, we found a consensus that a network consists of many „layers“:
  - Tensors are transformed using different „layers“, e.g.
    - Activation Layers (Relu, Sigmoid)
    - Fully Connected Layers
    - Convolution Layers
    - …

<<Interface>>
Layer

Todo…

An example network

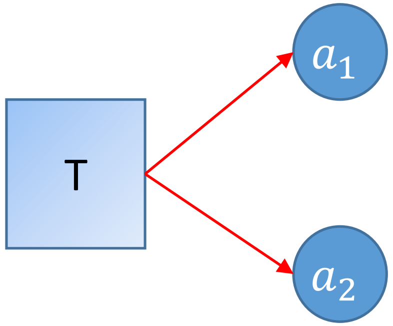
- Let us consider the following network

1. An input T is transformed to a vector of 2 numbers (Inputlayer)
2. A Fully Connected Layer Maps the network input to 3 numbers
3. A sigmoid Layer calculates a nonlinearity on that
4. A second Fully Connected Layer maps these values back to 2 values
5. We „normalize“ these values (also called „scores“) using a softmax activation
6. We calculate the loss using the „Cross-Entropy“ Loss function

An example network

- Let us consider the following network

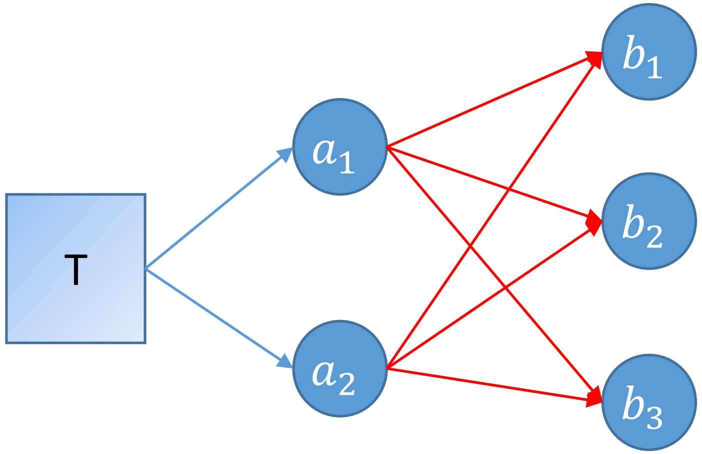
1. An input T is transformed to a vector of 2 numbers (Inputlayer)
2. A Fully Connected Layer Maps the network input to 3 numbers
3. A sigmoid Layer calculates a nonlinearity on that
4. A second Fully Connected Layer maps these values back to 2 values
5. We „normalize“ these values (also called „scores“) using a softmax activation
6. We calculate the loss using the „Cross-Entropy“ Loss function



An example network

- Let us consider the following network

1. An input T is transformed to a vector of 2 numbers (Inputlayer)
2. A Fully Connected Layer Maps the network input to 3 numbers
3. A sigmoid Layer calculates a nonlinearity on that
4. A second Fully Connected Layer maps these values back to 2 values
5. We „normalize“ these values (also called „scores“) using a softmax activation
6. We calculate the loss using the „Cross-Entropy“ Loss function

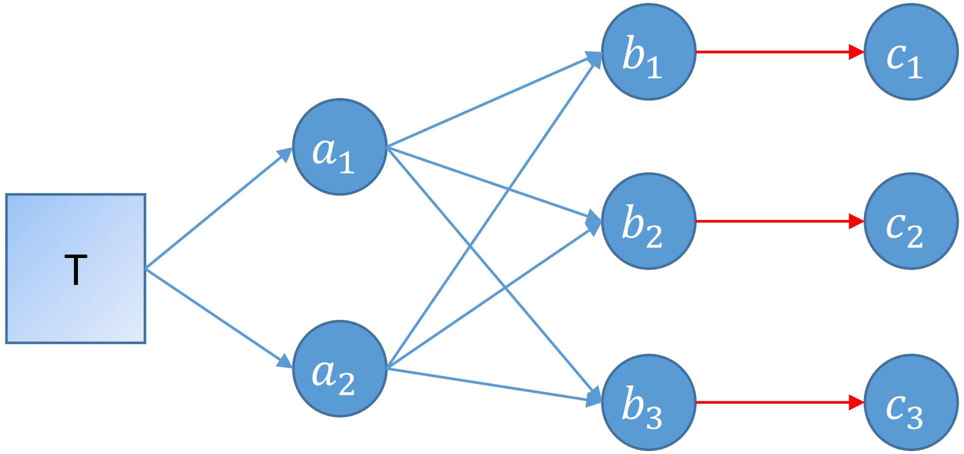


$$\vec{b} = \vec{a} \cdot W_1 + \overrightarrow{\mathrm{bias}_1}$$

An example network

- Let us consider the following network

1. An input T is transformed to a vector of 2 numbers (Inputlayer)
2. A Fully Connected Layer Maps the network input to 3 numbers
3. A sigmoid Layer calculates a nonlinearity on that
4. A second Fully Connected Layer maps these values back to 2 values
5. We „normalize“ these values (also called „scores“) using a softmax activation
6. We calculate the loss using the „Cross-Entropy“ Loss function



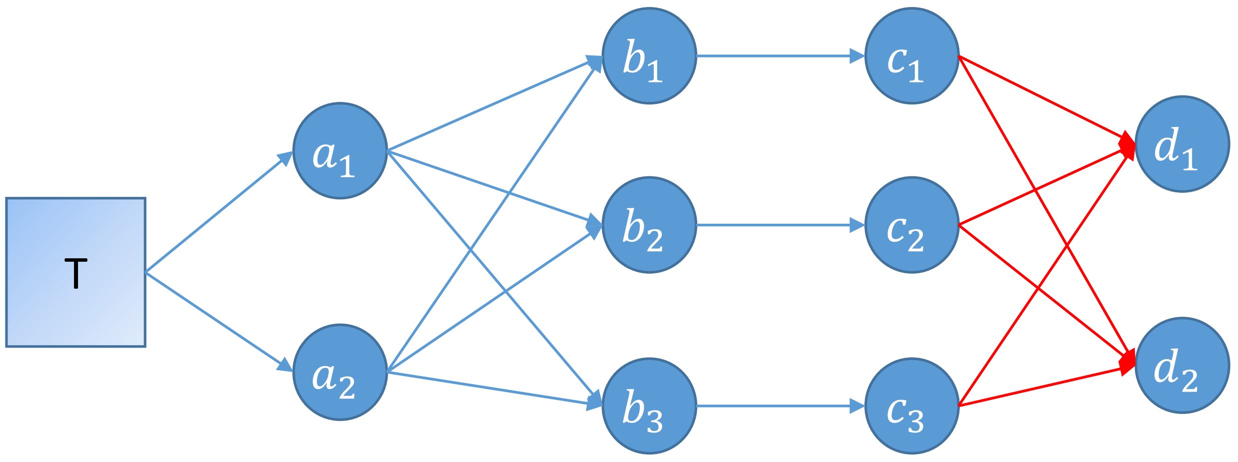
$$\vec{c} = \sigma(\vec{b})$$

$$\sigma(b_i) = \frac{1}{1 + \exp(-b_i)}$$

An example network

- Let us consider the following network

1. An input T is transformed to a vector of 2 numbers (Inputlayer)
2. A Fully Connected Layer Maps the network input to 3 numbers
3. A sigmoid Layer calculates a nonlinearity on that
4. A second Fully Connected Layer maps these values back to 2 values
5. We „normalize“ these values (also called „scores“) using a softmax activation
6. We calculate the loss using the „Cross-Entropy“ Loss function

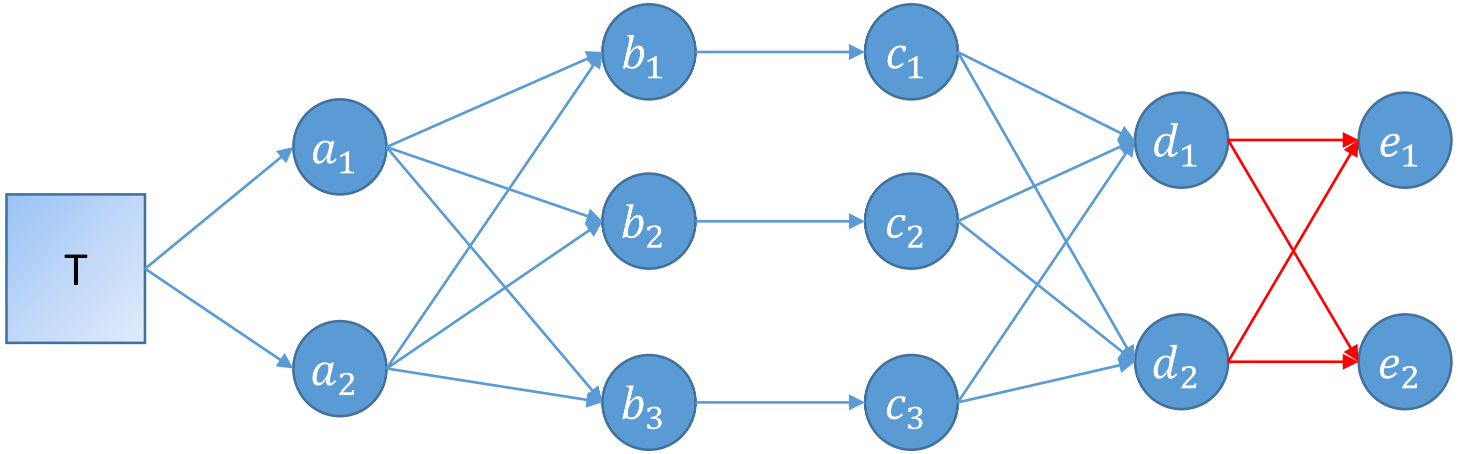


$$\vec{d} = \vec{c} \cdot W_2 + \overrightarrow{\text{bias}_2}$$

An example network

- Let us consider the following network

1. An input T is transformed to a vector of 2 numbers (Inputlayer)
2. A Fully Connected Layer Maps the network input to 3 numbers
3. A sigmoid Layer calculates a nonlinearity on that
4. A second Fully Connected Layer maps these values back to 2 values
5. We „normalize“ these values (also called „scores“) using a softmax activation
6. We calculate the loss using the „Cross-Entropy“ Loss function



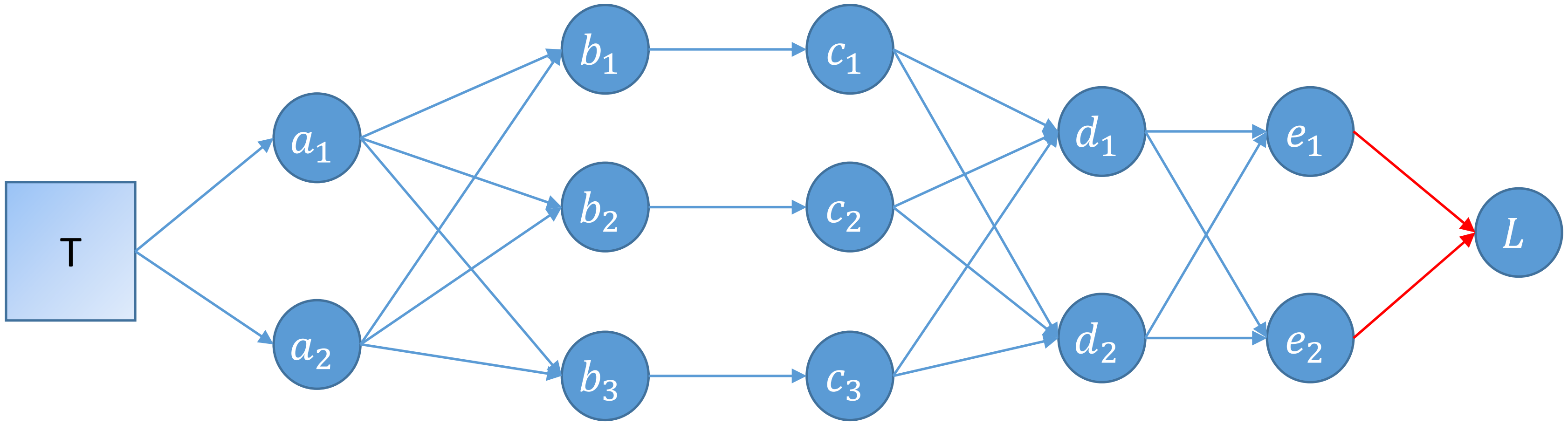
$$\vec{e} = \mathrm{softmax}(\vec{d})$$

$$\mathrm{softmax}(d_i) = \frac{\exp(d_i)}{\sum_j \exp(d_j)}$$

An example network

- Let us consider the following network

1. An input T is transformed to a vector of 2 numbers (Inputlayer)
2. A Fully Connected Layer Maps the network input to 3 numbers
3. A sigmoid Layer calculates a nonlinearity on that
4. A second Fully Connected Layer maps these values back to 2 values
5. We „normalize“ these values (also called „scores“) using a softmax activation
6. We calculate the loss using the „Cross-Entropy“ Loss function

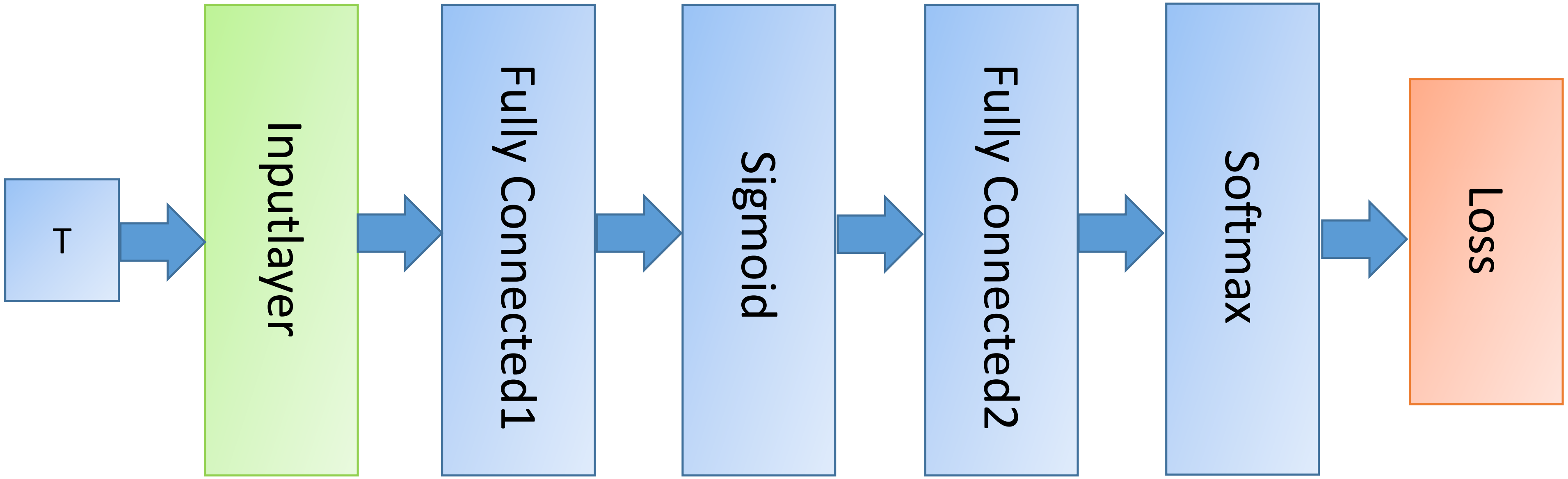


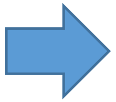
$$L = -\sum_i t_i \cdot \log(e_i)$$

t_i ist sehr oft einfach 0 oder 1.
Das gewünschte Label

An example network

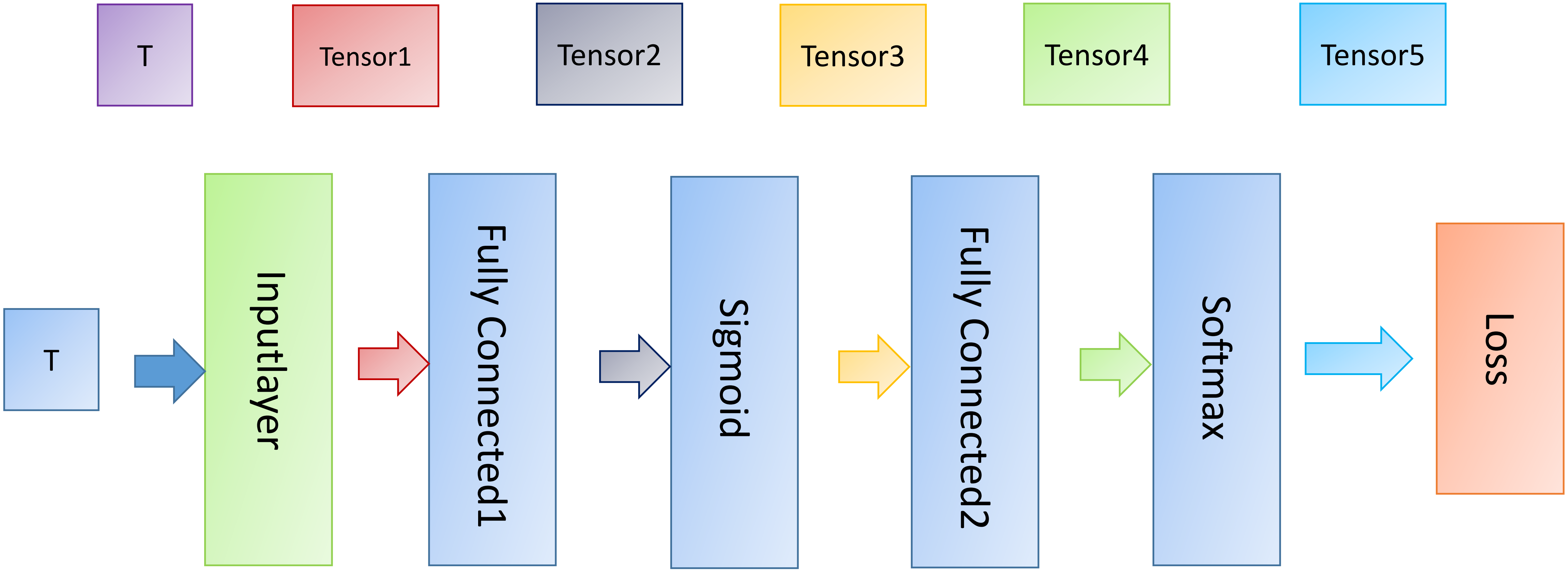
- We could draw it even more simplistic




Denotes the „forward pass“

An example network

- Forward



The Network - Layers

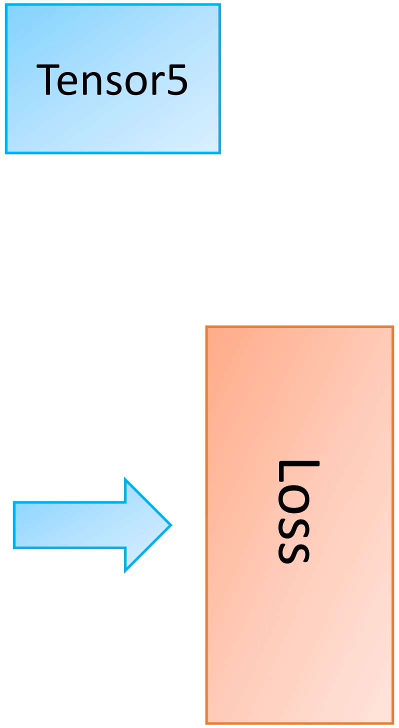
- A layer needs to be able to „forward“ incoming tensor(s)
    - inTensors: A list of incoming tensors (in this lecture always of size 1)
    - outTensors: A list of outgoing tensors
- The main purpose of the forward pass is to fill in the elements of the **outTensors**, while having access to the elements in **inTensors**

<<Interface>> Layer
forward(inTensors: List<Tensor>, outTensors: List<Tensor>) **Todo**

Remark –Forward Pass

- Why cant I just return the output values?
  - Of course you can! But this means one of the two things:
    1. You either have preallocated storage of the tensors in your layer
       ➔ which is pretty bad if you wanna run it in parallel
    2. You reallocate the same amount of storage every time your invoke the method
       ➔Pretty slow!

The Network - Layers



- At the end of the forward pass, we calculated the elements of all tensors until the Loss function is reached

- If we do not train then we can just return the last tensor
- However, if we are training we are not done yet…

Gradient Descent

- A neural network is a function with (usually many) parameters θ

$$\text{NN} = Loss(f_4(f_3(f_2(f_1(x, \theta_1), \theta_2), \theta_3), \theta_4), t)$$

- Optimizing such a function means we have to calculate the derivative and set it to $\vec{0}$
- The derivative with respect to multiple variables results in a vector, with one entry per variable. This vector is called the **gradient** $\vec{\nabla}$

Gradient: Example

- Given the function: $f(x_1, x_2) = \sin(x_1) + x_1 \cdot x_2$

- We calculate the derivative with respect to x_1:

$$\frac{\partial f}{\partial x_1} = \cos(x_1) + x_2$$

- And the derivate with respect to x_2:

$$\frac{\partial f}{\partial x_2} = x_1$$

- So the gradient $\overrightarrow{\nabla_f} = \begin{bmatrix} \cos(x_1) + x_2 \\ x_1 \end{bmatrix}$

Gradient Descent

- Given a function f , an arbitrary initial guess of our variables $\vec{x}$, a magical learning rate η, and the gradient $\overrightarrow{\nabla_f}$, gradient descent is an algorithm which finds us an local minimum for our function.

- This is done by the following easy steps:

1. Evaluate the gradient at $\vec{x}$
2. If the gradient is not $\vec{0}$ (or not close enough to it)
3. Update $\vec{x} = \vec{x} - \eta\overrightarrow{\nabla_f(x)}$
4. Repeat

Gradient Descent: Example

- Given:

1. a function $f(\mathrm{x}) = x^2 + 2x - 2$
2. The gradient $\overrightarrow{\nabla_f} = [2x + 2]$
3. a magical learning rate $\eta = 0.5$
4. an arbitrary initial guess of our variables $\vec{x} = (x) = (0)$

1. Iteration

Current: $f(0) = -2$
Evaluate $\overrightarrow{\nabla_f}(\vec{x}) = 2$
$\vec{x} = 0 - 0.5 \cdot 2 = -1$

2. Iteration

Current: $f(-1) = 1 - 2 - 2 = -3$
Evaluate $\overrightarrow{\nabla_f}(\vec{x}) = 0$

➔ Finished

Gradient Descent: Explained

- Why does this even work?
- Start at a random position $\overrightarrow{x_0}$
- Goal is to find a direction $\vec{p}$ $\text{(unit length)}$and a stepwidth $\eta (> 0)$ that minimizes our function

$$\min_{\eta > 0} f(x_0 + \eta\vec{p}\,)$$

- Start with the Taylor-Approximation around $\overrightarrow{x_0}$

$$f(x_0 + \eta\,\vec{p}) = f(x_0) + \eta\vec{p}^T\,\vec{\nabla} f(x_0) + \epsilon^2 + \cdots$$

Gradient Descent: Explained

- Start with the Taylor-Approximation around $\overrightarrow{x_0}$

$$f(x_0 + \eta\, \vec{p}) = f(x_0) + \eta \vec{p}^T\, \vec{\nabla} f(x_0) + \epsilon^2 + \cdots$$

- In this approximation around $\overrightarrow{x_0}$, we want to progress to the smallest value, so we want:

$$argmin(f)_{\vec{p}}$$

- It is minimized, when $\vec{p}^T\, \vec{\nabla} f(x_0)$ is minimized

$$\rightarrow \vec{p}^T\, \vec{\nabla} f(x_0) = ||\vec{p}^T|| \cdot \left\|\vec{\nabla} f(x_0)\right\| \cos(\phi) = \underbrace{\left\|\vec{\nabla} f(x_0)\right\| \cos(\phi)}$$

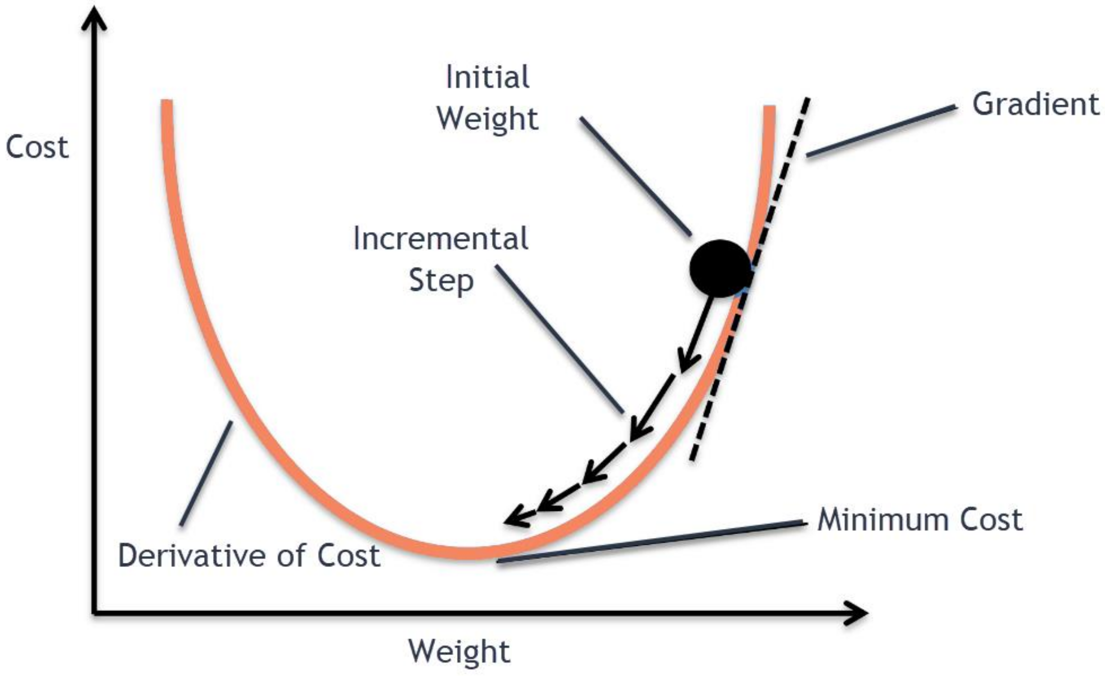
$$\rightarrow \vec{p} = -\frac{\vec{\nabla} f(x_0)}{\left\|\vec{\nabla} f(x_0)\right\|}$$

This is min if the cosine is -1

Gradient Descent: Explained

$$\rightarrow \vec{p} = -\frac{\vec{\nabla} f(x_0)}{\|\vec{\nabla} f(x_0)\|}$$

➔ This is just the direction of the negative gradient



Gradient Descent and Neural Networks

- A neural network is a function with (usually many) parameters θ

$$\text{NN} = Loss(f_4(f_3(f_2(f_1(x, \theta_1), \theta_2), \theta_3), \theta_4), t)$$

- We can fit these parameters to our data by guessing initial values θ_0 and update them iteratively in the negative direction of the gradient ∇_θ

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla_\theta L(f_\theta(x_i), t_i)$$

- This means we need to (efficiently) calculate ∇_θ:

$$\nabla_\theta = \begin{bmatrix} \frac{\partial L}{\theta_0} \\ \frac{\partial L}{\theta_1} \\ \dots \end{bmatrix}$$

Backpropagation

- This is done using a „dynamic program“ called „Backpropagation“

> **Dynamic Program in a nutshell**: Calculate the solution of a bigger problem, using the solution of a smaller programm at the cost of some intermediatery variables

- Other well known algorithms using this concept:
  - ???

Backpropagation

- This is done using a „dynamic program“ called „Backpropagation“

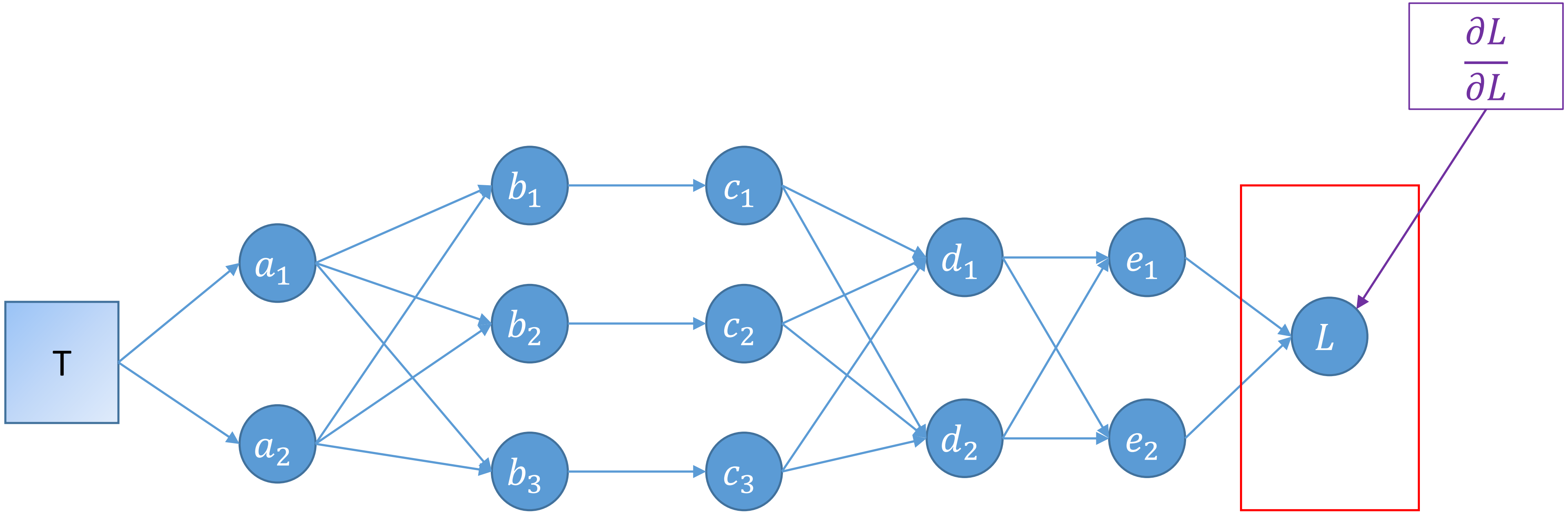
> **Dynamic Program in a nutshell**: Calculate the solution of a bigger problem, using the solution of a smaller programm at the cost of some intermediatery variables

- Other well known algorithms using this concept:
  - Viterbi Algorithm
  - CKY Algorithm
  - Forward Backward Algorithm

Backpropagation

- This is done using a „dynamic program“ called „Backpropagation“

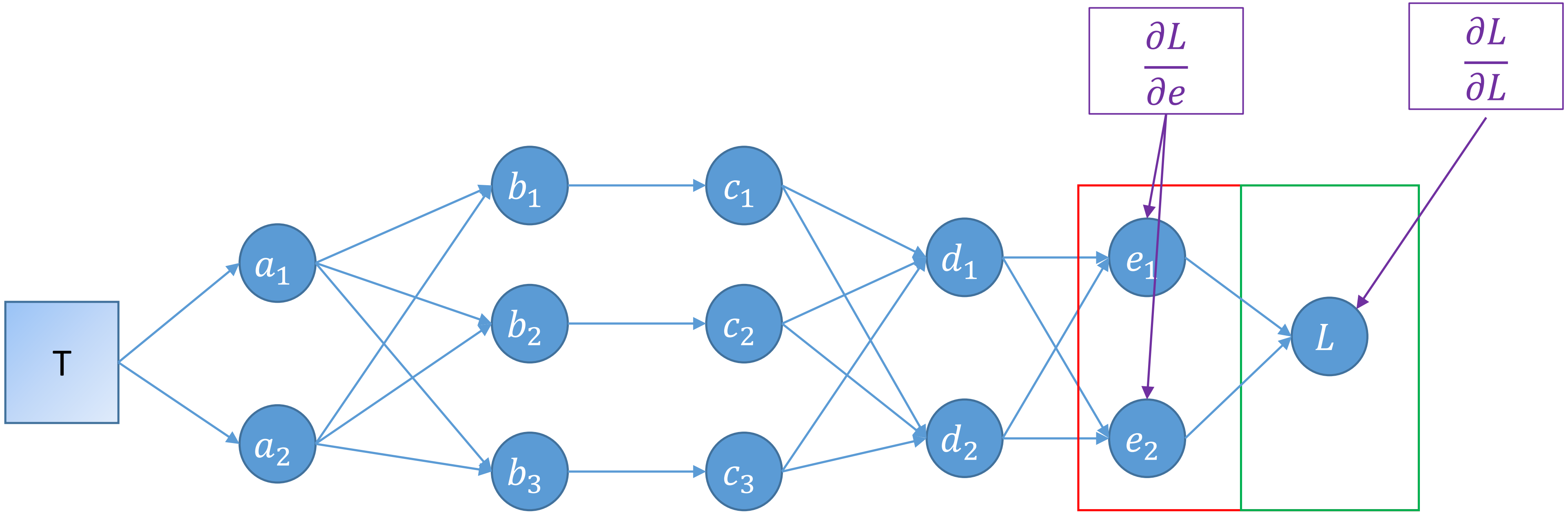
<u>Dynamic Program in a nutshell</u>: Calculate the solution of a bigger problem, using the solution of a smaller programm at the cost of some intermediatery variables



Backpropagation

- This is done using a „dynamic program“ called „Backpropagation“

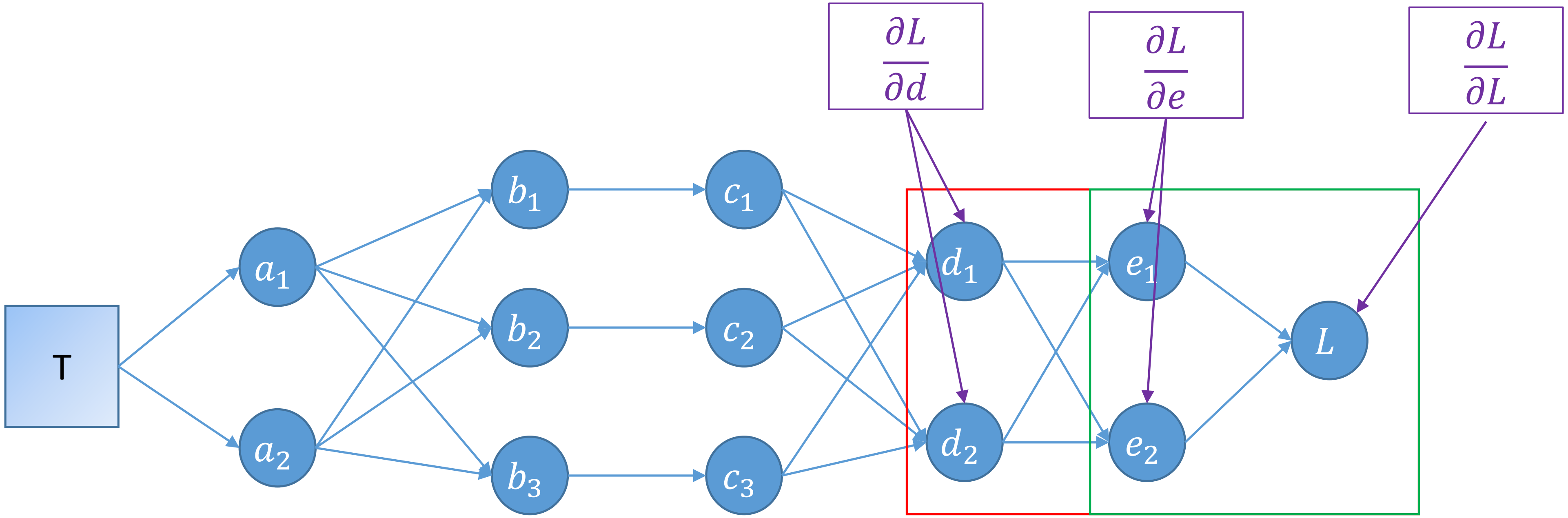
Dynamic Program in a nutshell: Calculate the solution of a bigger problem, using the solution of a smaller programm at the cost of some intermediatery variables



Backpropagation

- This is done using a „dynamic program“ called „Backpropagation“

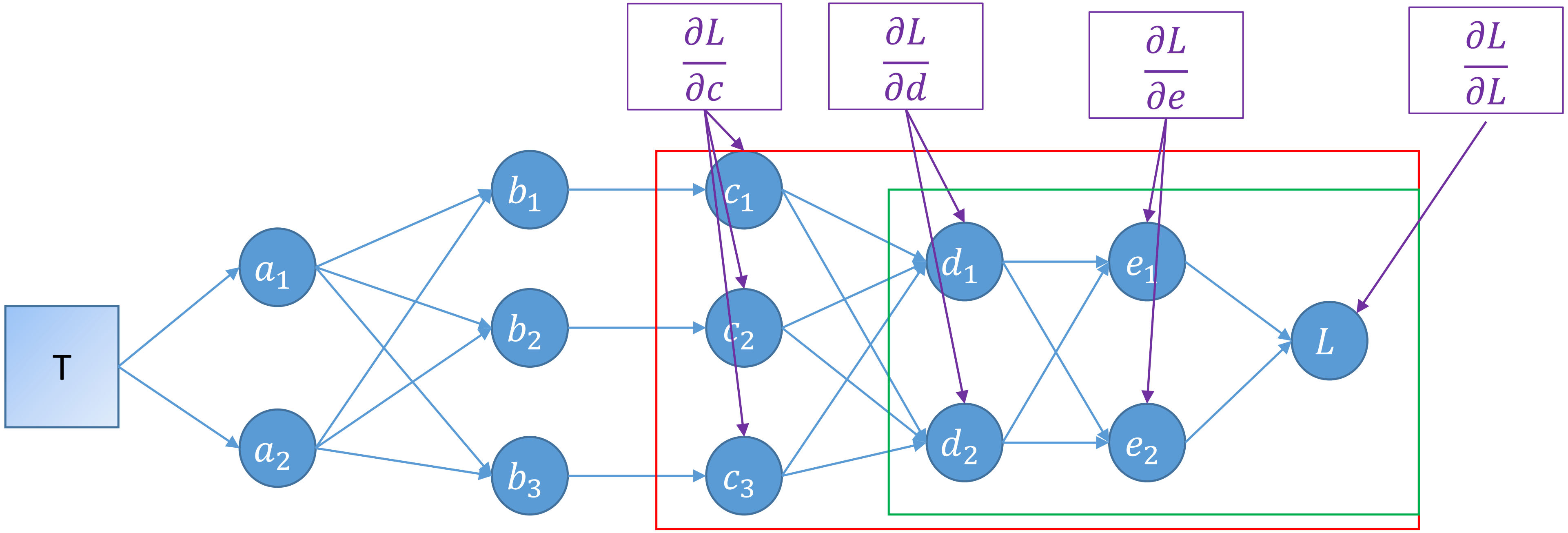
<u>Dynamic Program in a nutshell</u>: Calculate the solution of a bigger problem, using the solution of a smaller programm at the cost of some intermediatery variables



Backpropagation

- This is done using a „dynamic program“ called „Backpropagation“

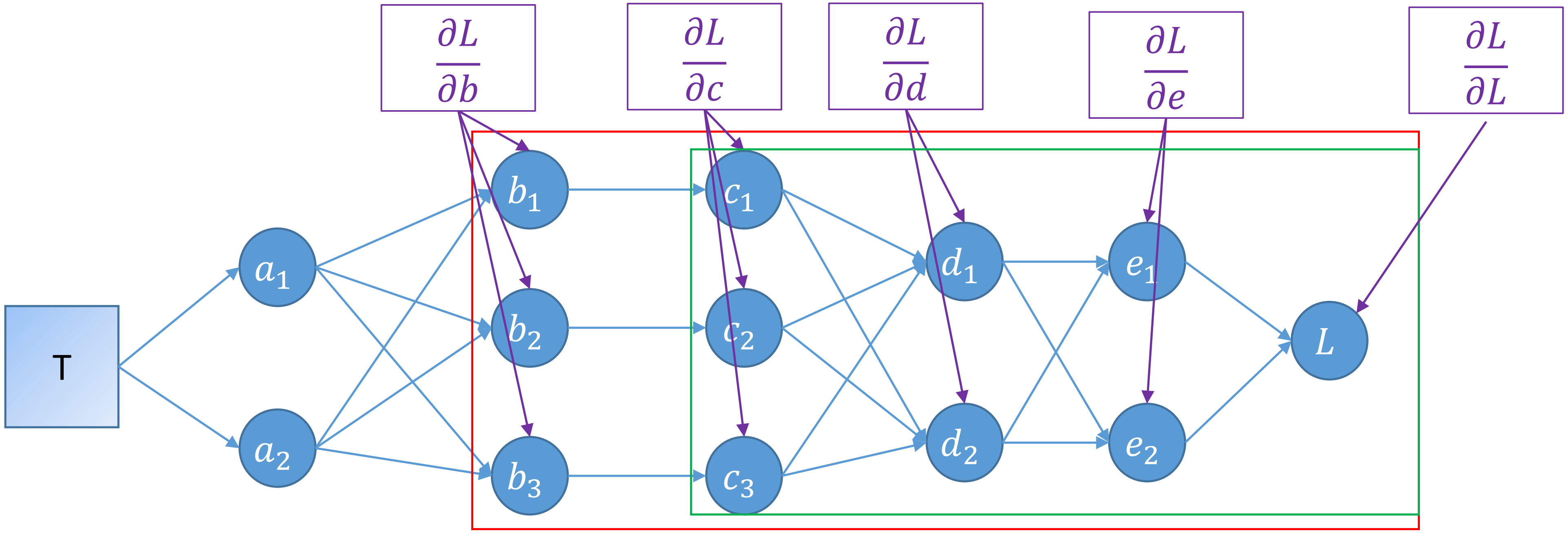
<u>Dynamic Program in a nutshell</u>: Calculate the solution of a bigger problem, using the solution of a smaller programm at the cost of some intermediatery variables



Backpropagation

- This is done using a „dynamic program" called „Backpropagation"

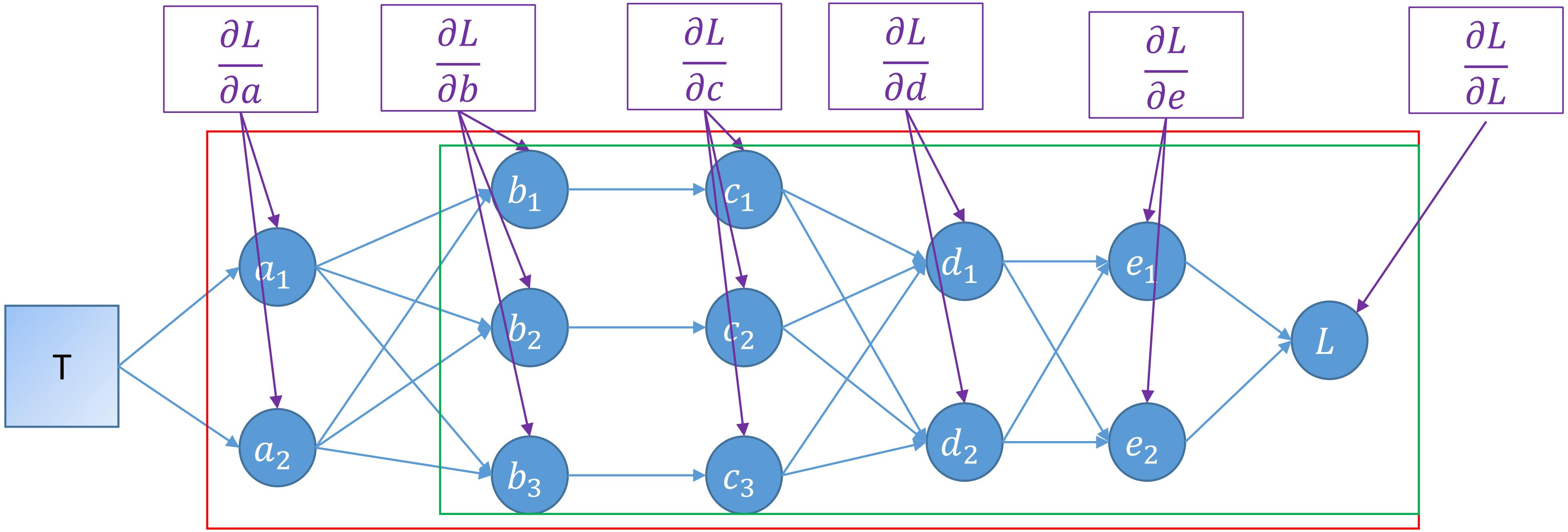
Dynamic Program in a nutshell: Calculate the solution of a bigger problem, using the solution of a smaller programm at the cost of some intermediatery variables



Backpropagation

- This is done using a „dynamic program“ called „Backpropagation“

Dynamic Program in a nutshell: Calculate the solution of a bigger problem, using the solution of a smaller programm at the cost of some intermediatery variables



Backpropagation

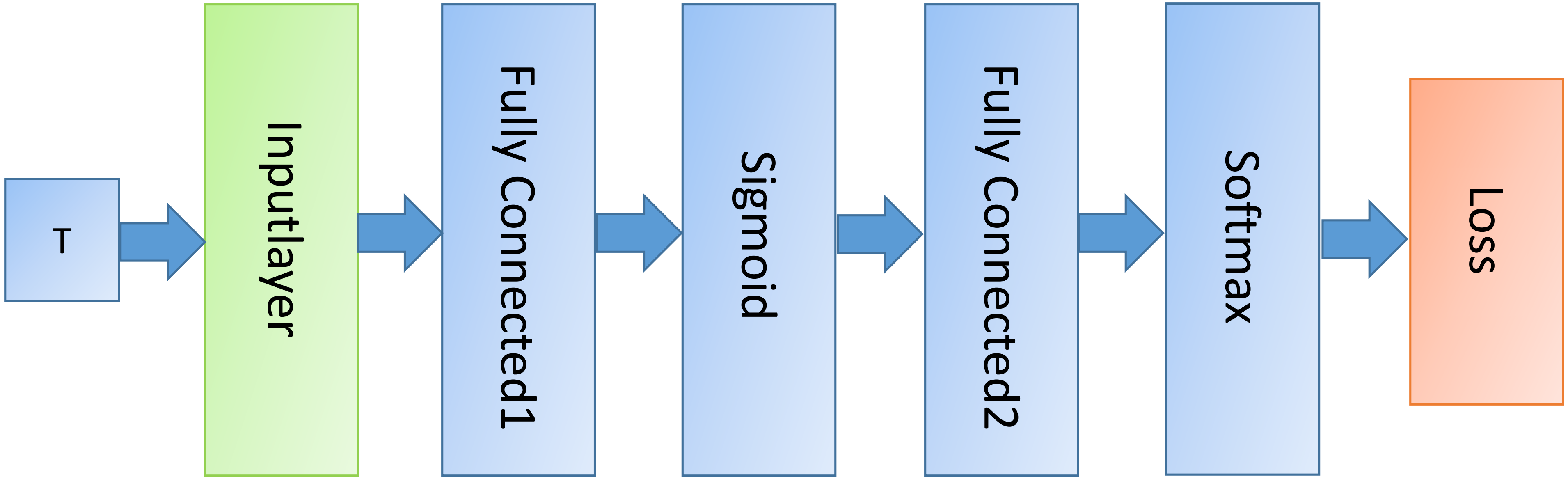
- What we want:
- A procedure to (efficiently) calculate ∇_θ:

$$\nabla_\theta = \begin{bmatrix} \frac{\partial L}{\theta_0} \\ \frac{\partial L}{\theta_1} \\ \dots \end{bmatrix}$$

- What we got (so far) a procedure to efficiently calculate some intermediary variables that we have no idea yet why we need them

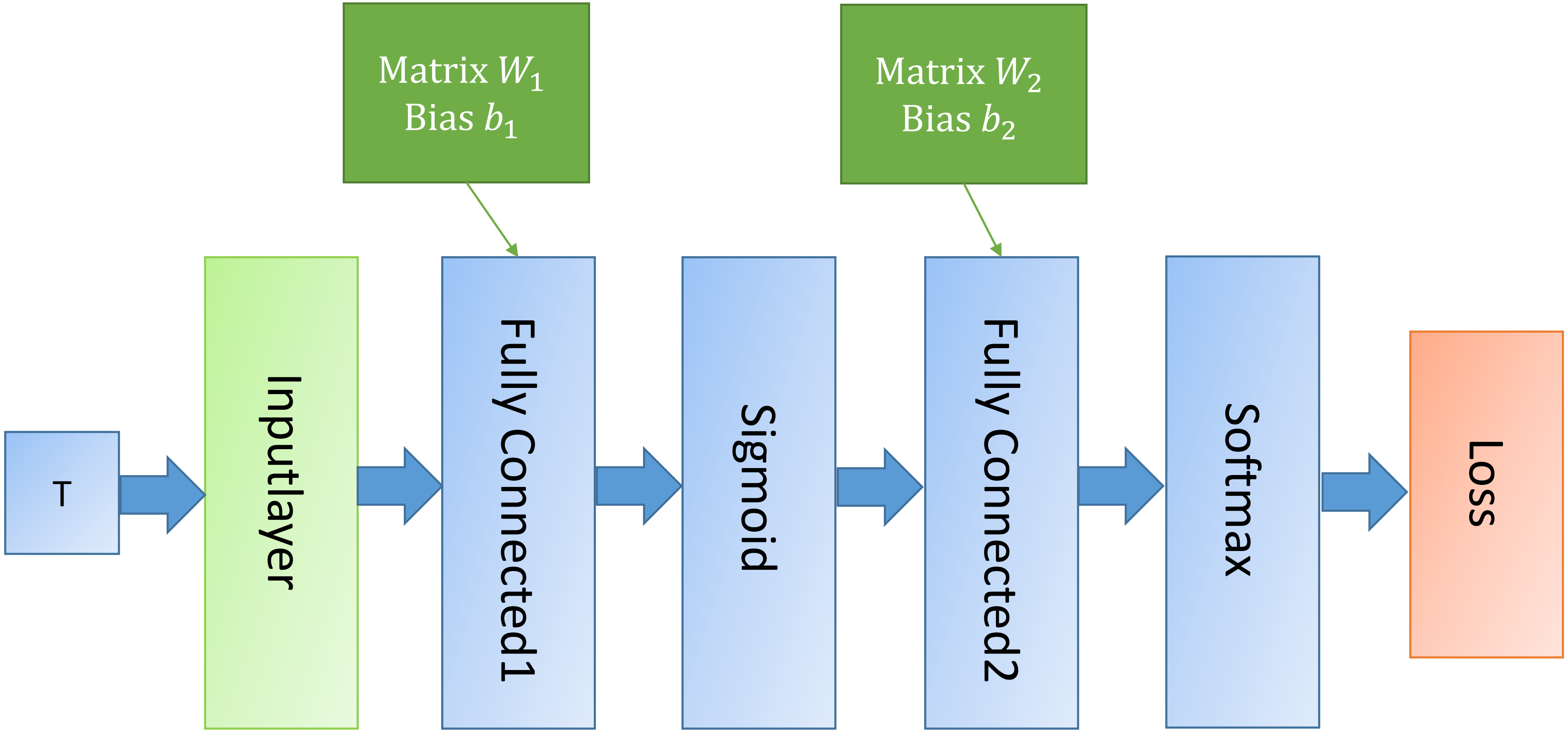
Backpropagation

- Where`s ~~Waldo~~ θ ?



Backpropagation

- Where`s ~~Waldo~~ θ ?



The Network - Layers

- A layer needs to be able to „backward“ outgoing tensor(s)
  - inTensors: A list of incoming tensors
  - outTensors: A list of outgoing tensors
- The main purpose of the backward pass is to fill in the deltas of the **inTensors**, while having access to the deltas in **outTensors** as well as the elements of the **inTensors**

➔ **Where to store the deltas?**

<<Interface>>
Layer

forward(inTensors: List<Tensor>, outTensors: List<Tensor>)

backward(outTensors: List<Tensor>, inTensors: List<Tensor>)

Todo

Data structures: Tensor - Revisited

- In order to store the delta variables, we need to introduce another array in the Tensor
- Make sure that this variable is initialized „lazy“ since it would otherwise double the amount of RAM that is required for your application

```
//and the deltas, init as zero array,
val deltas: FloatArray by lazy {
    (0 until size).map { 0.0f }.toFloatArray()
}
```

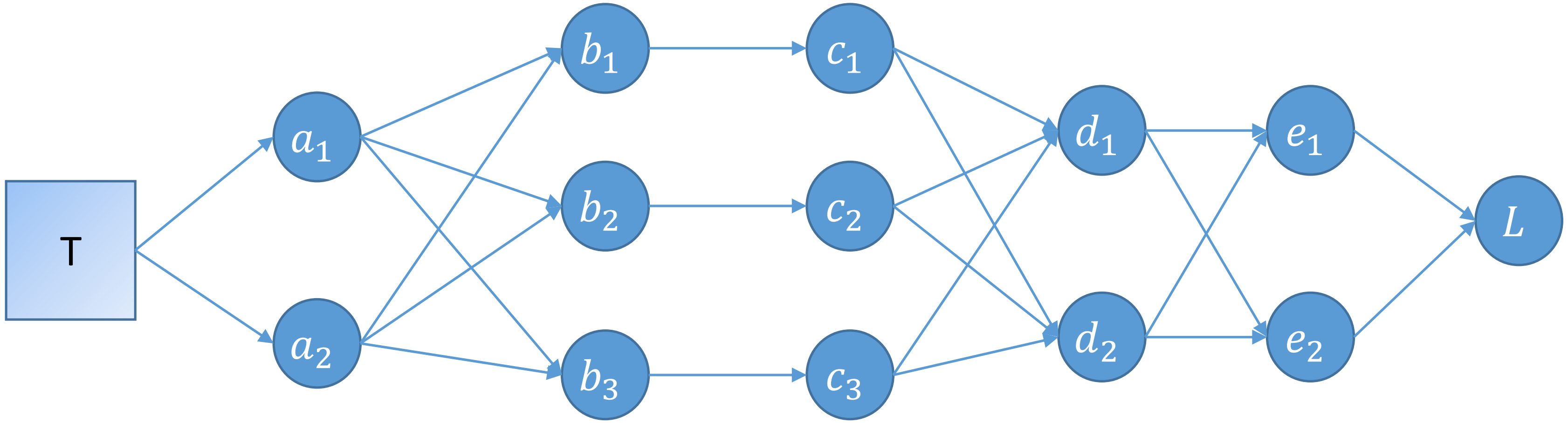
Tensor

elements: FloatArray
shape: Shape

Deltas: lazy FloatArray

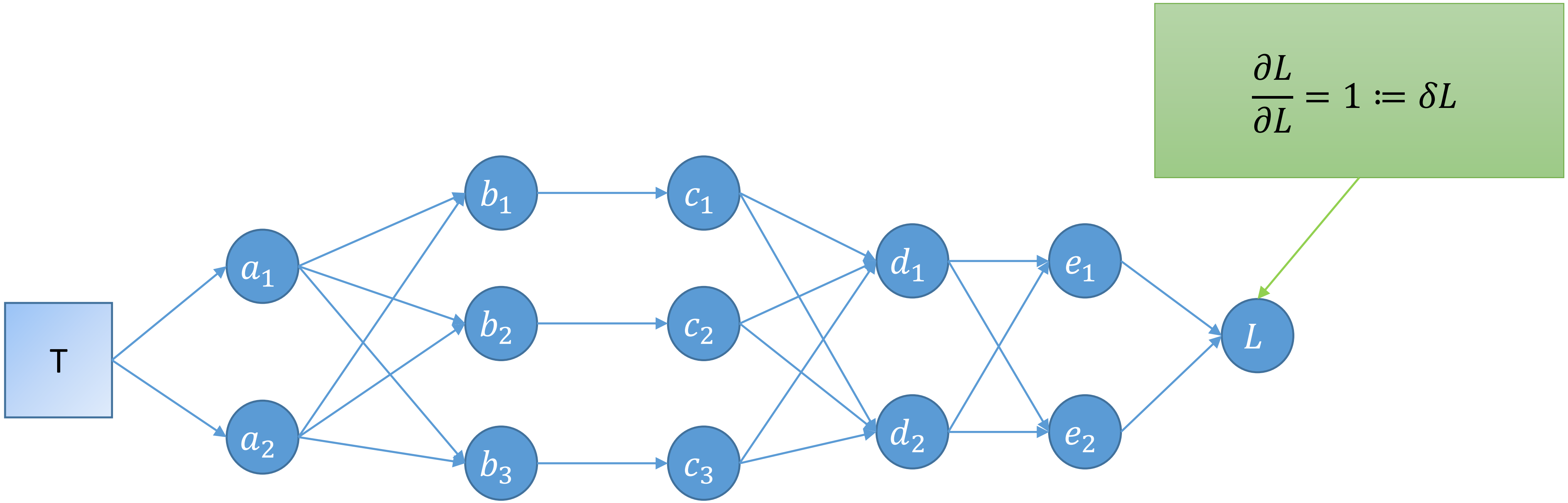
Backward -example (single data point)

- Let us now determine all intermediary variables, for this running example



Backward -example (single data point)

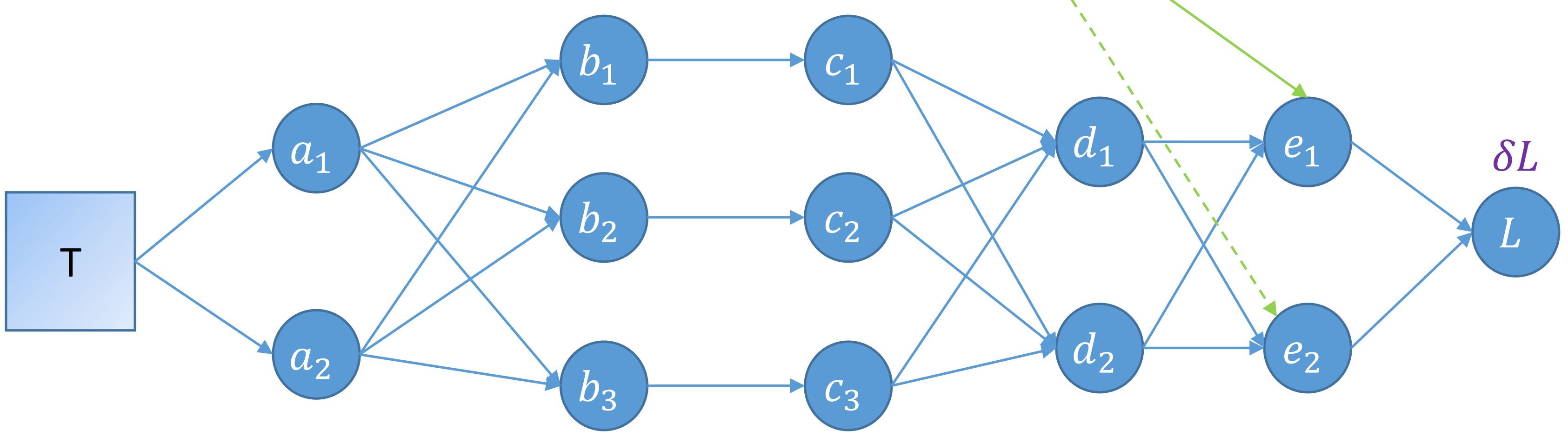
- Let us now determine all intermediary variables, for this running example



Backward -example (single data point)

- Let us now determine all intermediary variables, for this running example

$$\frac{\partial L}{\partial e_1} = \frac{\partial L}{\partial L} \cdot \frac{\partial L}{\partial e_1} = \delta L \cdot \frac{\partial L}{\partial e_1} := \delta e_1$$



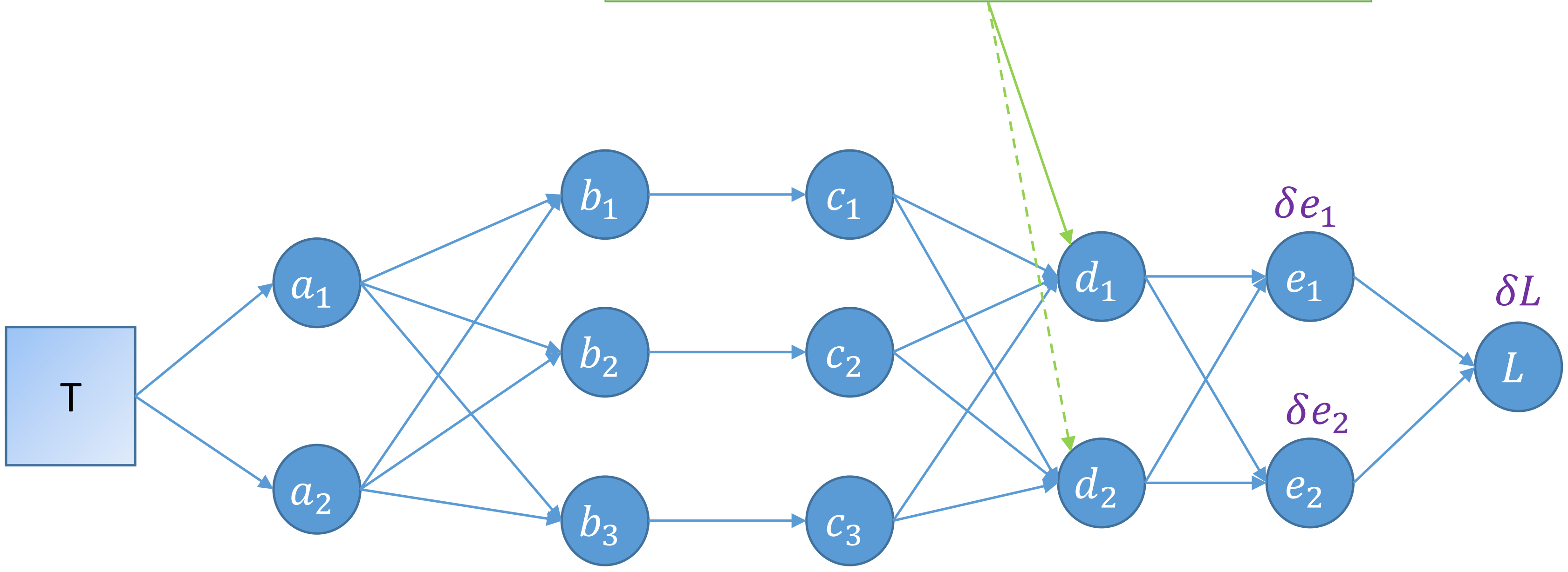
$$L = -\sum_i t_i \cdot \log(e_i)$$

$$\rightarrow \frac{\partial L}{\partial e_1} = -\frac{t_1}{e_1}$$

Backward -example (single data point)

- Let us now determine all intermediary variables, for this running example

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d} = \delta e \cdot \frac{\partial e}{\partial d} := \delta d$$



$$\mathrm{softmax}(d_i) = \frac{\exp(d_i)}{\sum_j \exp(d_j)} = e_i$$

Backward -example (single data point)

Determine $\frac{\partial e}{\partial d}$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d} = \delta e \cdot \frac{\partial e}{\partial d} := \delta d$$

$$\text{softmax}(d_i) = \frac{\exp(d_i)}{\sum_j \exp(d_j)}$$

$$\vec{e} = \begin{bmatrix} e_1 & e_2 \end{bmatrix} = \begin{bmatrix} \text{softmax}(d_1) & \text{softmax}(d_2) \end{bmatrix} \rightarrow \frac{\partial e}{\partial d} = \begin{bmatrix} \frac{\partial e_1}{\partial d_1} & \frac{\partial e_1}{\partial d_2} \\ \frac{\partial e_2}{\partial d_1} & \frac{\partial e_2}{\partial d_2} \end{bmatrix}$$

Backward -example (single data point)

Determine $\frac{\partial e}{\partial d}$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d} = \delta e \cdot \frac{\partial e}{\partial d} := \delta d$$

$$\text{softmax}(d_i) = \frac{\exp(d_i)}{\sum_j \exp(d_j)}$$

$$\vec{e} = [e_1 \quad e_2] = [\text{softmax}(d_1) \quad \text{softmax}(d_2)] \rightarrow \frac{\partial e}{\partial d} = \begin{bmatrix} \frac{\partial e_1}{\partial d_1} & \frac{\partial e_1}{\partial d_2} \\ \frac{\partial e_2}{\partial d_1} & \frac{\partial e_2}{\partial d_2} \end{bmatrix}$$

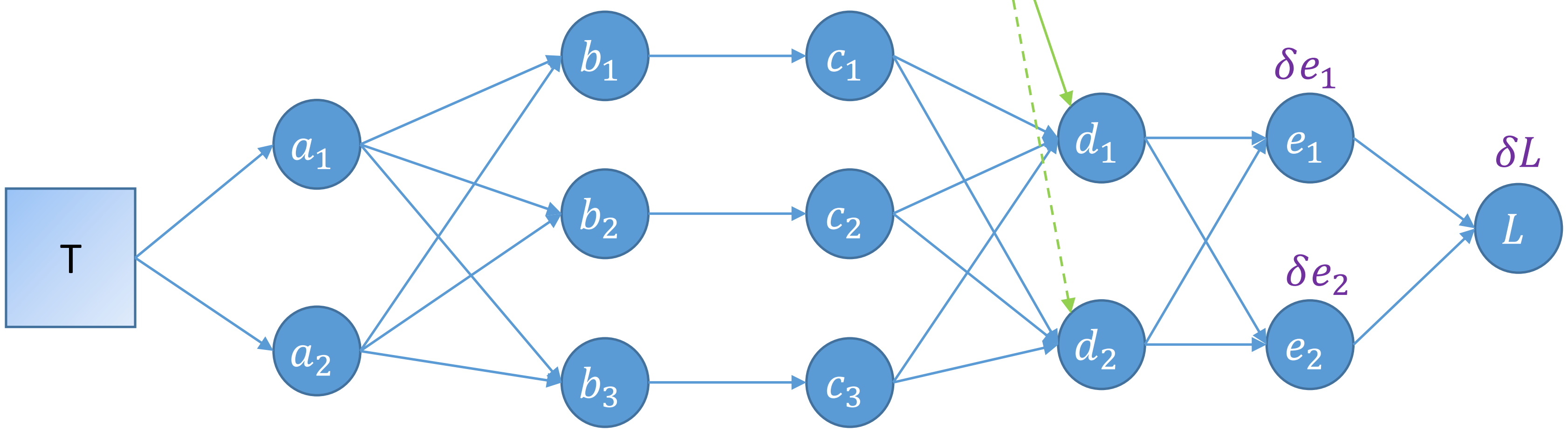
$$\rightarrow \frac{\partial e}{\partial d} = \begin{bmatrix} sm(d_1)\,(1 - sm(d_1)) & -sm(d_2)\, sm(d_1) \\ -sm(d_1)\, sm(d_2) & sm(d_2)\,(1 - sm(d_2)) \end{bmatrix}$$

$\rightarrow$ in general $\frac{\partial e_i}{\partial d_j} = sm(d_i) \cdot (\delta_{ij} - sm(d_j))$ with δ_{ij} being the Kronecker delta

Backward -example (single data point)

- Let us now determine all intermediary variables, for this running example

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d} = \delta e \cdot \frac{\partial e}{\partial d} := \delta d$$



$$\frac{\partial e}{\partial d} = \begin{bmatrix} \frac{\partial e_1}{\partial d_1} & \frac{\partial e_2}{\partial d_1} \\ \frac{\partial e_1}{\partial d_2} & \frac{\partial e_2}{\partial d_2} \end{bmatrix}$$

$$\rightarrow \frac{\partial L}{\partial d} = [\delta e_1 \; \delta e_2] \cdot \begin{bmatrix} \frac{\partial e_1}{\partial d_1} & \frac{\partial e_2}{\partial d_1} \\ \frac{\partial e_1}{\partial d_2} & \frac{\partial e_2}{\partial d_2} \end{bmatrix}$$

Backward: Why all this hassle?

- Why does this hold? $\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d}$

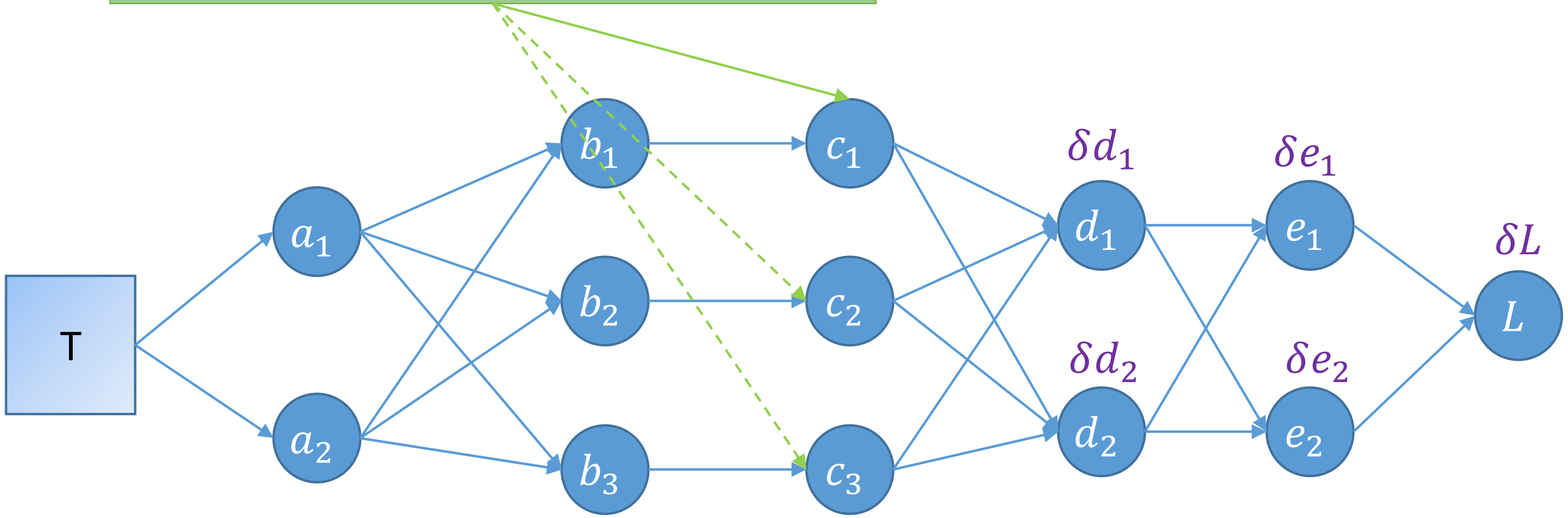
- Imagine we got a function: $\mathrm{f} = L(d(c(b(a))))$
- We now want to calculate the partial derivatives:

- $\frac{\partial L}{\partial d} = \frac{\partial L}{\partial d}$
- $\frac{\partial L}{\partial c} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial c}$
- $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial c} \cdot \frac{\partial c}{\partial b}$
- $\frac{\partial L}{\partial a} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial c} \cdot \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial a}$

Backward -example (single data point)

- Let us now determine all intermediary variables, for this running example

$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial c} = \delta d \cdot \frac{\partial d}{\partial c} := \delta c$$



$$\vec{d} = \vec{c} \cdot W_2 + \overrightarrow{\text{bias}_2}$$

$\frac{\overrightarrow{\partial d}}{\partial c}$ is the derivative of the vector-matrix multiplication, with regard to the vector

Derivative of vector-matrix product

- We will now derive the derivative of the vector-matrix product ($\vec{y} = \vec{x} \cdot W$), with respect to the vector

- Using the previous example:

$$\vec{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \qquad W = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}$$

- $y_1 = x_1 \cdot w_{11} + x_2 \cdot w_{12} + x_3 \cdot w_{13}$
- $y_2 = x_1 \cdot w_{21} + x_2 \cdot w_{22} + x_3 \cdot w_{23}$
- $y_3 = x_1 \cdot w_{31} + x_2 \cdot w_{32} + x_3 \cdot w_{33}$

Derivative of vector-matrix product

- We will now derive the derivative of the vector-matrix product ($\vec{y} = \vec{x} \cdot W$), with respect to the vector

$$\vec{y} = \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \qquad W = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}$$

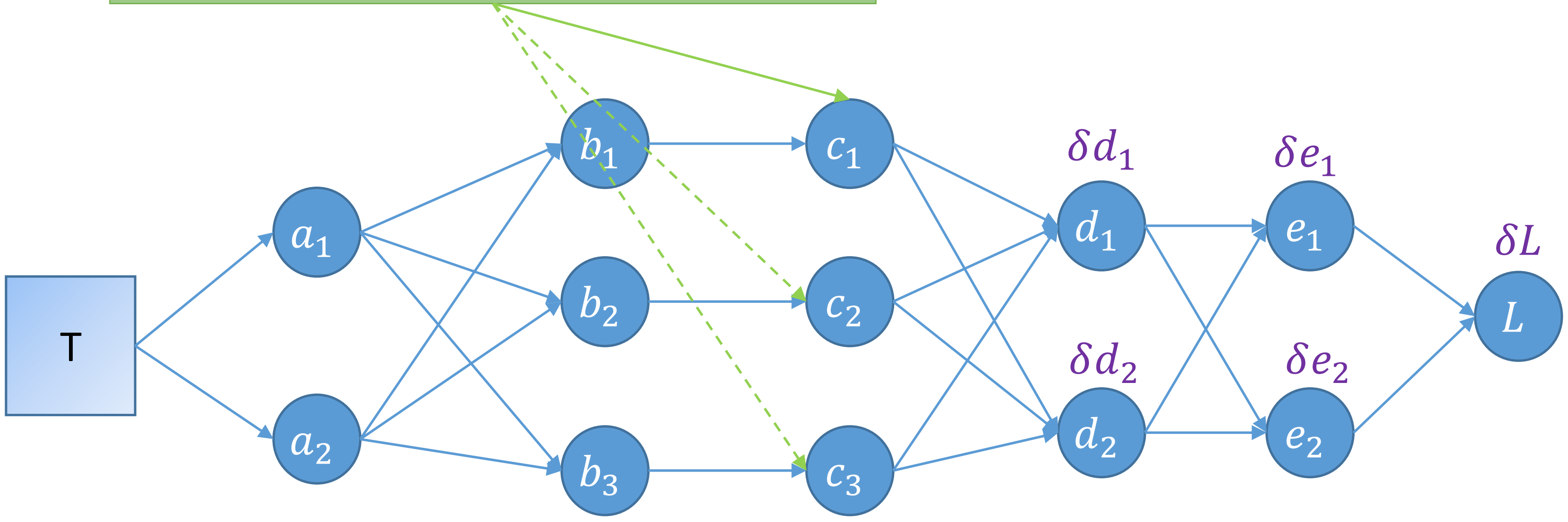
$$\boxed{\begin{aligned} y_1 &= x_1 \cdot w_{11} + x_2 \cdot w_{12} + x_3 \cdot w_{13} \\ y_2 &= x_1 \cdot w_{21} + x_2 \cdot w_{22} + x_3 \cdot w_{23} \\ y_3 &= x_1 \cdot w_{31} + x_2 \cdot w_{32} + x_3 \cdot w_{33} \end{aligned}}$$

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \frac{\partial y}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \\ \frac{\partial y_3}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_3}{\partial x_3} \end{bmatrix} \rightarrow = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = W^T$$

Backward -example (single data point)

- Let us now determine all intermediary variables, for this running example

$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial d} \cdot \frac{\partial d}{\partial c} = \delta d \cdot \frac{\partial d}{\partial c} := \delta c$$



$$\vec{d} = \vec{c} \cdot W_2 + \overrightarrow{\text{bias}_2}$$

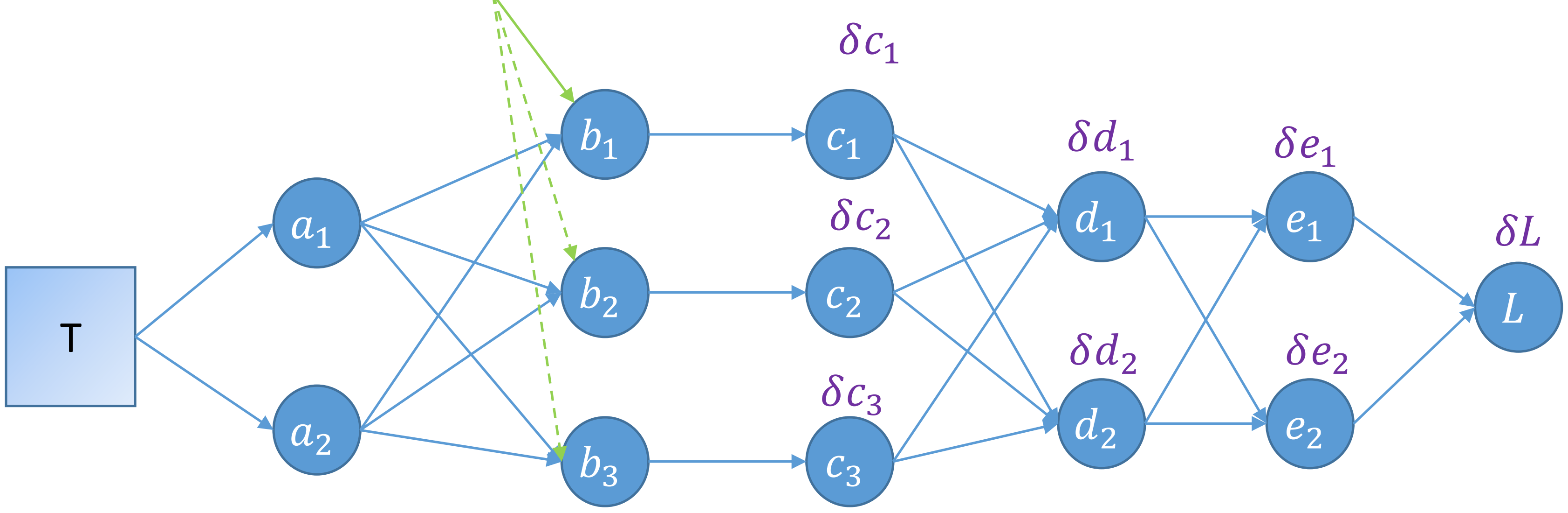
$$\frac{\overrightarrow{\partial d}}{\partial c} = W_2^T$$

$$\rightarrow \frac{\partial L}{\partial c} = \delta d \cdot W_2^T$$

Backward -example (single data point)

- Let us now determine all intermediary variables, for this running example

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial c} \cdot \frac{\partial c}{\partial b} = \delta c \cdot \frac{\partial c}{\partial b} := \delta b$$



$$\frac{\partial c}{\partial b} = \frac{\partial}{\partial b}\sigma(b) = \frac{\partial}{\partial b}\frac{1}{1+e^{-b}}$$

$=$ … chain rule and stuff

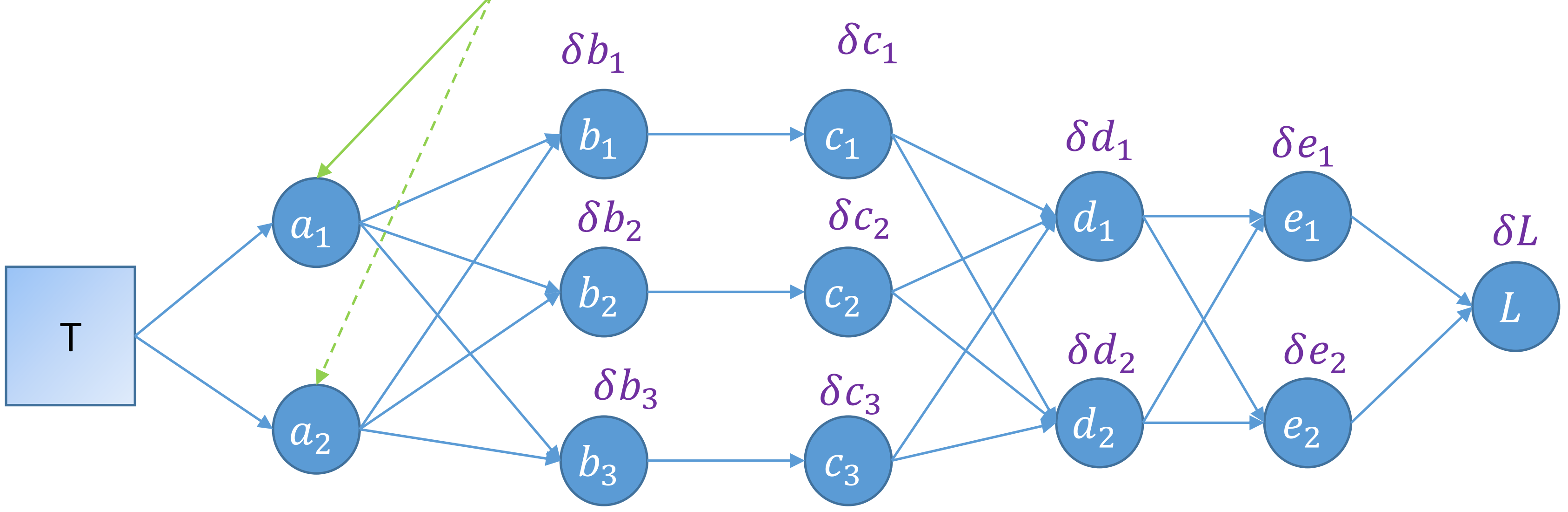
$$\Rightarrow \sigma`(b) = \sigma(b) \cdot (1 - \sigma(b))$$

Please do this as an exercise!

Backward -example (single data point)

- Let us now determine all intermediary variables, for this running example

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial a} = \delta b \cdot \frac{\partial b}{\partial a} := \delta a$$



$\frac{\partial b}{\partial a}$ is just another vector-matrix multiplication

➔ $\frac{\partial L}{\partial a} = \delta b \cdot W_1^T$

Backpropagation

- What we want:
- A procedure to (efficiently) calculate ∇_θ:

$$\nabla_\theta = \begin{bmatrix} \frac{\partial L}{\theta_0} \\ \frac{\partial L}{\theta_1} \\ \dots \end{bmatrix}$$

- What we got (so far) a procedure to efficiently calculate some intermediary variables that we have no idea yet why we need them

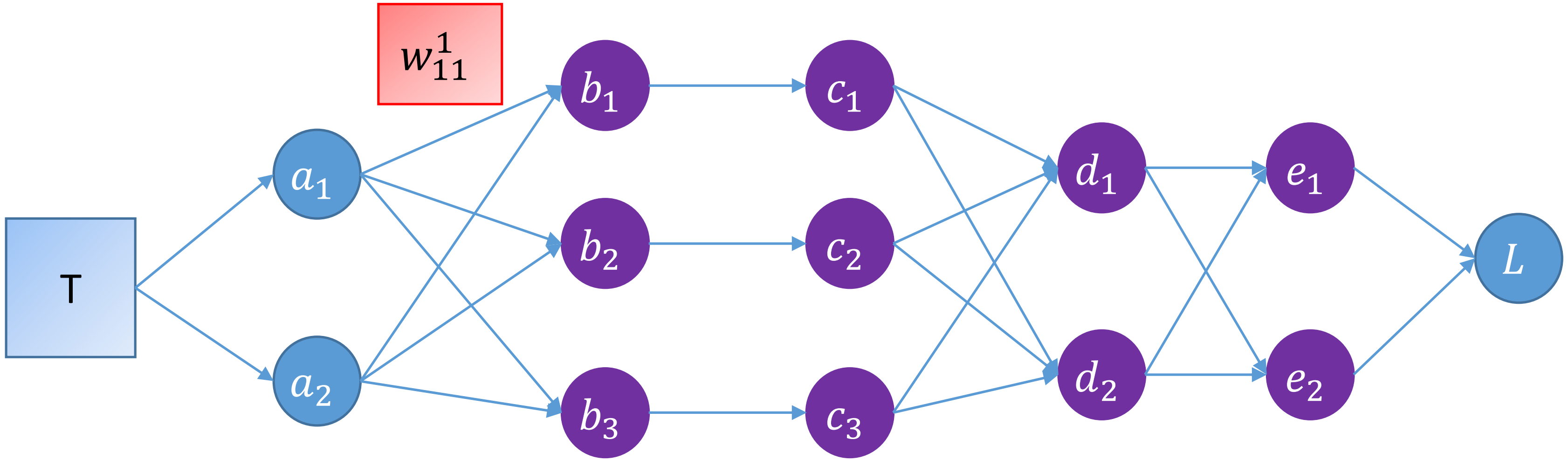
Parameter Update- Fully Connected

$$\frac{\partial L}{\partial w_{11}^1} = [\delta b_1, \delta b_2, \delta b_3] \cdot \frac{\partial b}{\partial w_{11}^1}$$

- We can now derive the final equation for our target parameter w_{11}^1:

$$\frac{\partial L}{\partial w_{11}^1} = \delta b_1 \cdot a_1$$

But why?



Parameter Update- Fully Connected, Explanation

$$b_1 = w_{11}^1 \cdot a_1 + w_{21}^1 \cdot a_2$$

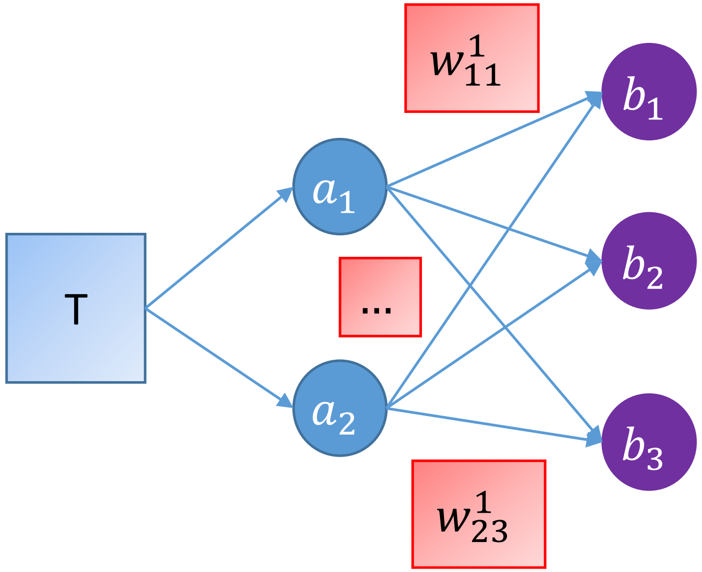
$$b_2 = w_{12}^1 \cdot a_1 + w_{22}^1 \cdot a_2$$

$$b_3 = w_{13}^1 \cdot a_1 + w_{23}^1 \cdot a_2$$

$$\frac{\partial L}{\partial w_{11}^1} = \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial w_{11}^1}$$

$$= \frac{\partial L}{\partial b} \cdot [\frac{\partial b_1}{\partial w_{11}^1}, \frac{\partial b_2}{\partial w_{11}^1}, \frac{\partial b_3}{\partial w_{11}^1}]$$

$$= [\delta b_1, \delta b_2, \delta b_3] \cdot [a_1, 0, 0] = \delta b_1 \cdot a_1$$



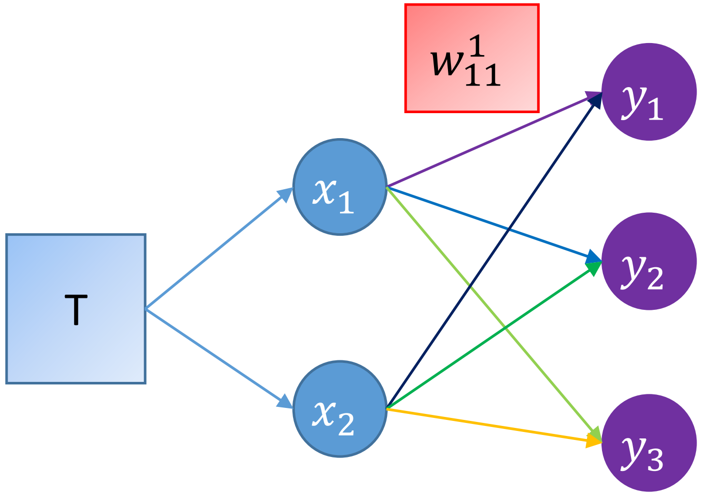
Derivation Derivative Vector-Matrix Multiplication

- We got $\vec{y} = \vec{x} \cdot W$ and we have already seen the partial derivate with respect to x.

- What we do now is basicly the same thing, but we form the derivative with respect to our matrix W

- $\frac{\partial y}{\partial W} = ?$, in our current example, the matrix has the form:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$



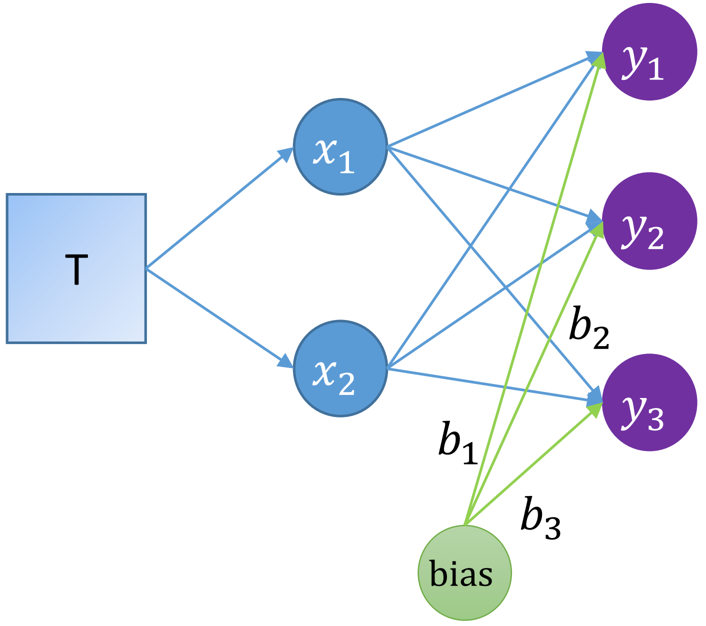
Repeating what we have done on the previous slide yields:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \delta y_1 x_1 & \delta y_2 x_1 & \delta y_3 x_1 \\ \delta y_1 x_2 & \delta y_2 x_2 & \delta y_3 x_2 \end{bmatrix} = \vec{x}^T \cdot \delta\vec{y} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} \delta y_1 & \delta y_2 & \delta y_3 \end{bmatrix}$$

What about the bias

- So far, we ignored the bias
- But a fully connected layer calculates: $\vec{y} = \vec{x}W + \vec{\mathrm{b}}$
- We are missing $\frac{\partial L}{\partial b}$
- Since the bias can be seen as an additional input into y, we can use the same formula we have seen on the derivation before, and therefore find:

$$\frac{\partial L}{\partial b_1} = \delta y_1 \qquad \text{and so on}$$



The Network - Layers

- A layer with parameters needs to be able to „calculate Delta Weights“ and it gets exactly the same inputs as the other methods!
  - inTensors: A list of incoming tensors
  - outTensors: A list of outgoing tensors

<<Interface>>
Layer

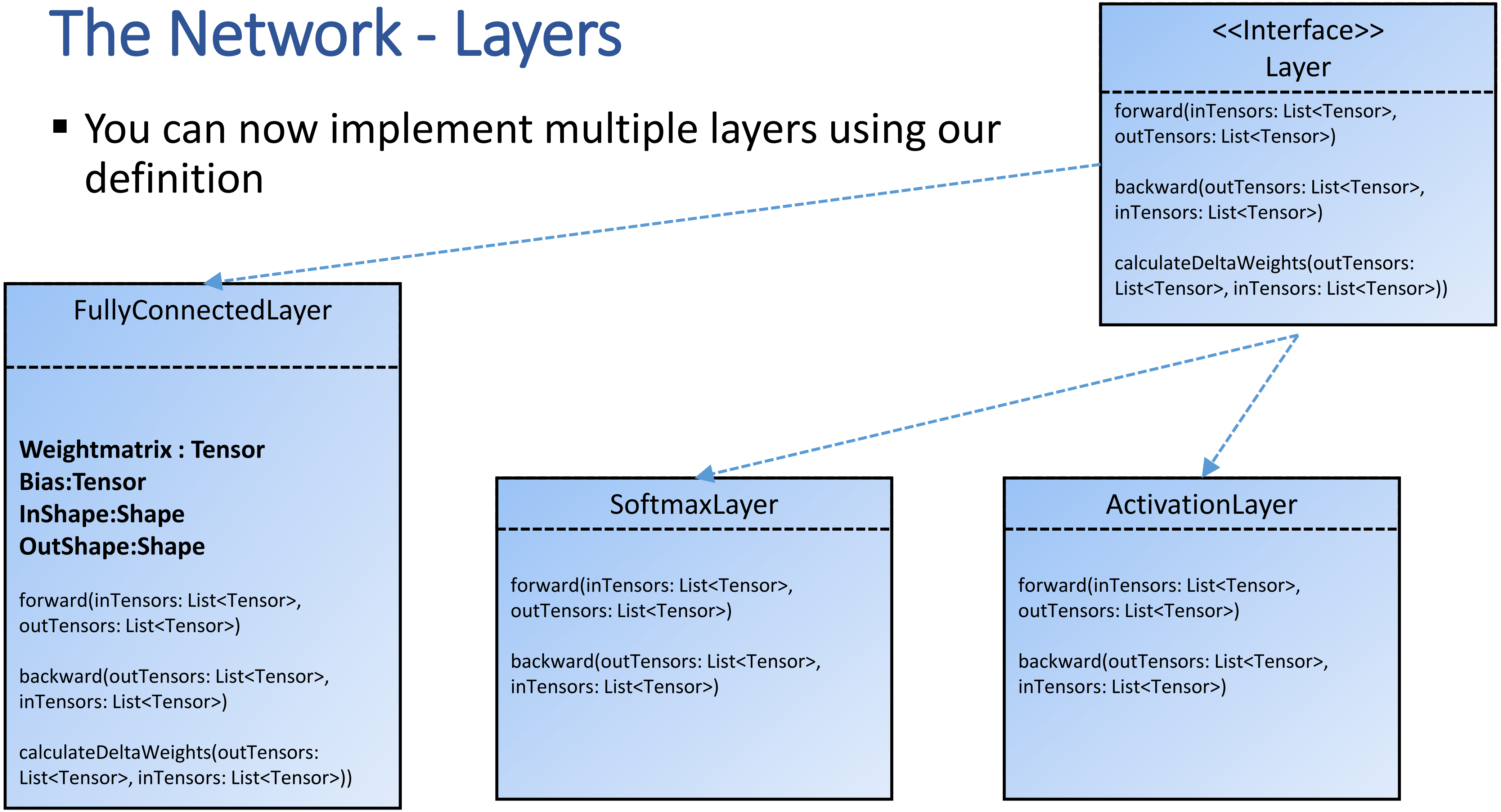
forward(inTensors: List<Tensor>, outTensors: List<Tensor>)

backward(outTensors: List<Tensor>, inTensors: List<Tensor>)

calculateDeltaWeights(outTensors: List<Tensor>, inTensors: List<Tensor>))

The Network - Layers

- You can now implement multiple layers using our definition



The Network - Layers

- If you paid close attention, then you might have noticed:

<u>Forward</u>: Use **elements of inTensors** (and parameters if available) to calculate **elements in outTensors**.

<u>Backward</u>: Use **deltas of outTensors** (and parameters if available) to calculate **deltas of inTensors**

<u>Parameter Update</u>: Use **elements of inTensors** and **deltas of outTensors** to **calculate delta Weights**

Cheat Sheet: Fully Connected

- Forward:

$$Y = X \cdot W + \text{bias}$$

- Backward:

$$\delta X = \delta Y \cdot W^T$$

- Parameter Update:

$$\frac{\partial L}{\partial W} = X^T \, \delta Y$$

$$\frac{\partial L}{\partial \text{bias}} = \delta Y$$

Cheat Sheet: Sigmoid Layer σ

- Forward:

$$Y = \sigma(X) = \frac{1}{1+e^{-X}} \qquad \text{(applied elementwise)}$$

- Backward:

$$\delta X = \left[\sigma(X) \cdot \left(1 - \sigma(X)\right)\right] \odot \delta Y$$

Cheat Sheet: Softmax Layer

- Forward:

$$Y = softmax(X) = \frac{e^{x_i}}{\sum_i e^{x_i}}$$

- Backward:

$$\delta X = \delta Y \cdot \begin{bmatrix} \frac{\partial x_1}{\partial y_1} & \cdots & \frac{\partial x_1}{\partial y_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_n}{\partial y_1} & \cdots & \frac{\partial x_n}{\partial y_n} \end{bmatrix}$$

Cheat Sheet: Cross Entropy

- Forward:

$$L = -\sum_i t_i \cdot \log(x_i)$$

- Backward:

$$\frac{\delta L}{\delta x_i} = -\frac{t_i}{x_i}$$

Cheat Sheet: Mean Squared Error

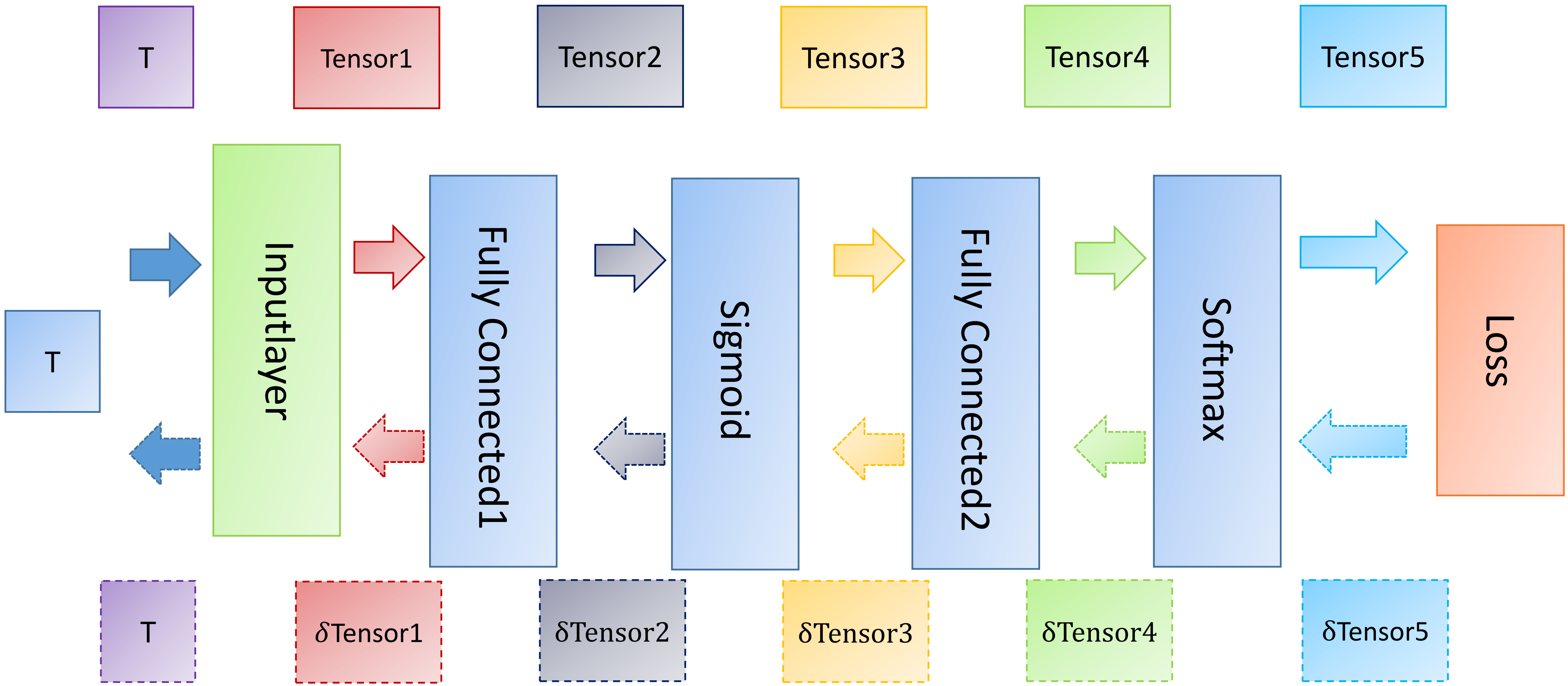
- Forward:

$$L = \sum_i \frac{1}{2}(x_i - t_i)^2$$

- Backward:

$$\frac{\delta L}{\delta x_i} = x_i - t_i$$

The big picture (quite literally)

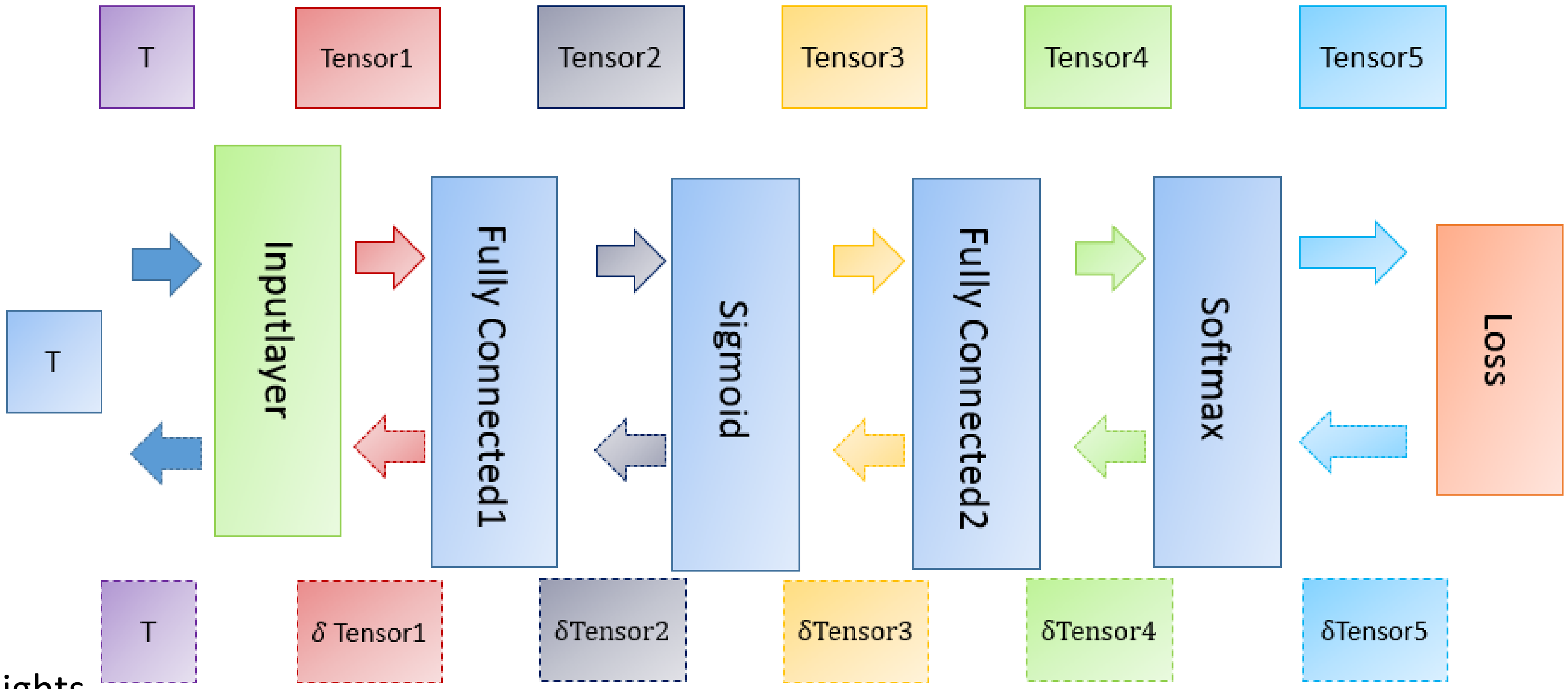


The function „backprop“

Phase1: forward

Phase2: backward

Phase3: store deltaWeights



The Network class

- The Network class takes an InputLayer, a Loss and a list (or varargs) of layers

Network

input: InputLayer<T>
layers: List<Layer>
\parameters:List<Tensor>
deltaParams: List<Tensor>
caches: <Your choice>

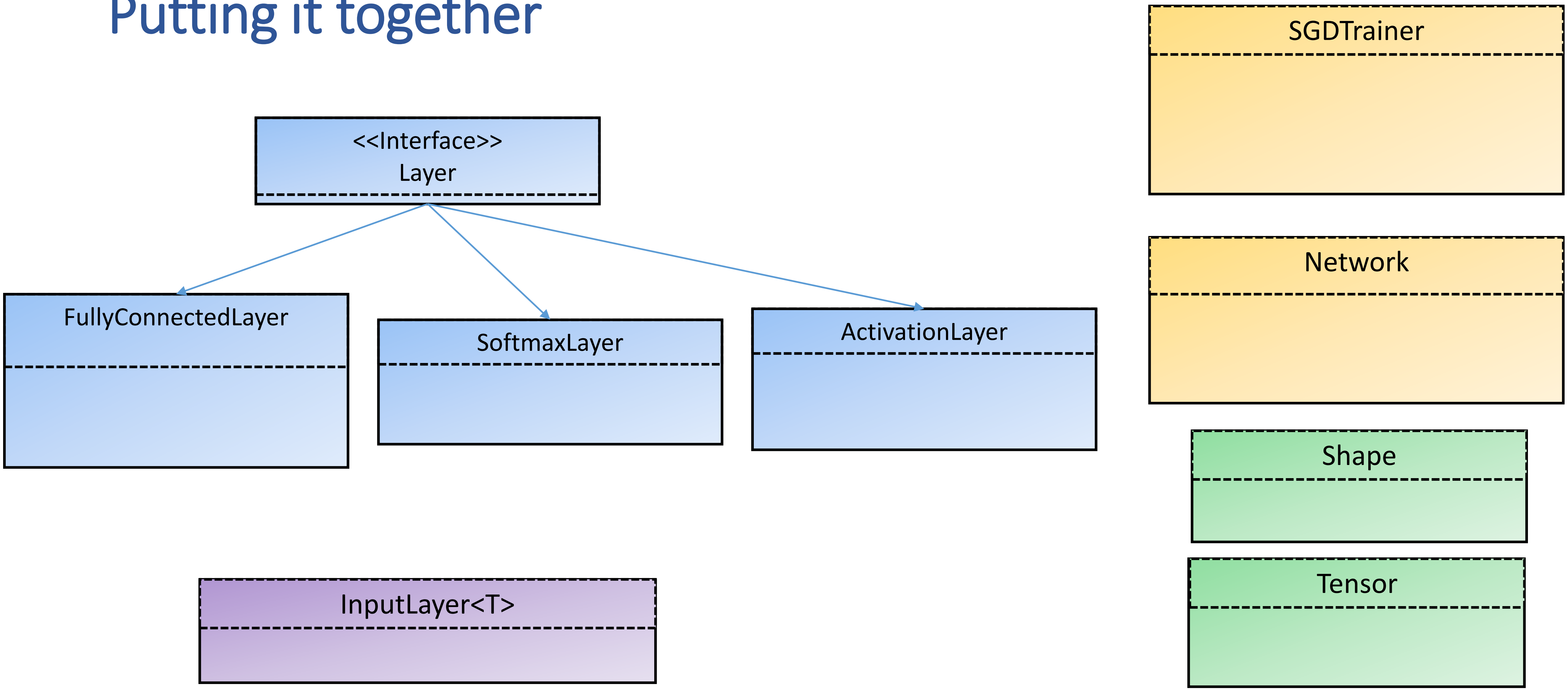
backprop(<Your Data here>)
forward()

The Trainer

- The training is delegated to a class SGDTrainer
- Ist only function is the optimize method which takes the data and the network
- The flavor is the algorithm which is used for parameter update:
  - Vanilla **Stochastic Gradient Descent**
  - Adagrad
  - Adam
  - RMSProp
  - …
  - See http://ruder.io/optimizing-gradient-descent/

SGDTrainer
batchSize:Int=1 learningRate:Float amountEpochs:Int Shuffle:Boolean=True updateMechanism: SGDFlavor optimize(network:Network,data)

Putting it together



Happy Coding!