Lecture 10:

ANN: Perceptron, Backpropagation, SGD

Markus Hohle

University California, Berkeley

Bayesian Data Analysis and Machine Learning for Physical Sciences

Course Map

Outline

- **Gradient Descent**
  - Vanilla
  - Learning Rate Schedule
  - L1 and L2
  - Momentum
  - AdaGrad
  - RMSProp
- **Perceptron**
- **Backpropagation**

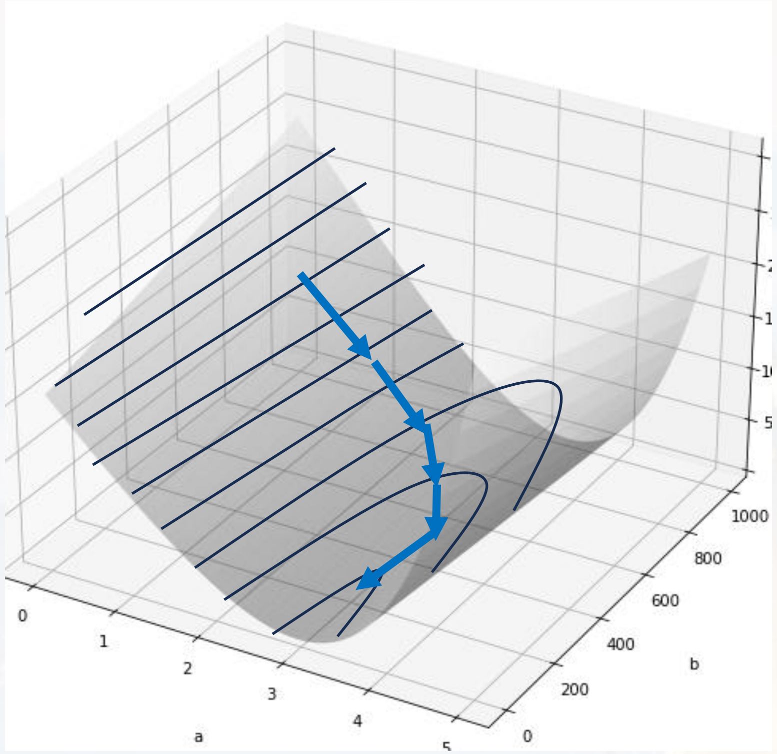
Outline

- Gradient Descent

- Vanilla
- Learning Rate Schedule
- L1 and L2
- Momentum
- AdaGrad
- RMSProp

- Perceptron

- Backpropagation



goal: find the (global) minimum of a loss function $\mathcal{L}$, e. g. $\mathcal{L} = \frac{1}{2}[y - \hat{y}]^2$

problem: no closed form of $\mathcal{L}$ as a function of the model parameters $\vec{x}$

idea: (stochastic) gradient descent

- $\epsilon > 0$: learning rate
- t: iteration
- κ: decay rate
- y: true value
- $\hat{y}$: model prediction

vanilla: initialize $\vec{x}_{t=0}$ usually $\mathcal{N}(0,1)$

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)$$

problem: ϵ too large → leads to oscillations
ϵ too small → takes too long to converge

one idea: decreasing learning rate

$$\epsilon(t) = \frac{\epsilon_{t=0}}{1 + \kappa\, t}$$

can also be a stepwise function (**learning rate schedule**)

$\epsilon > 0$:	learning rate
t:	iteration
κ:	decay rate
y:	true value
$\hat{y}$:	model prediction

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)$$

AdaptiveGradient:

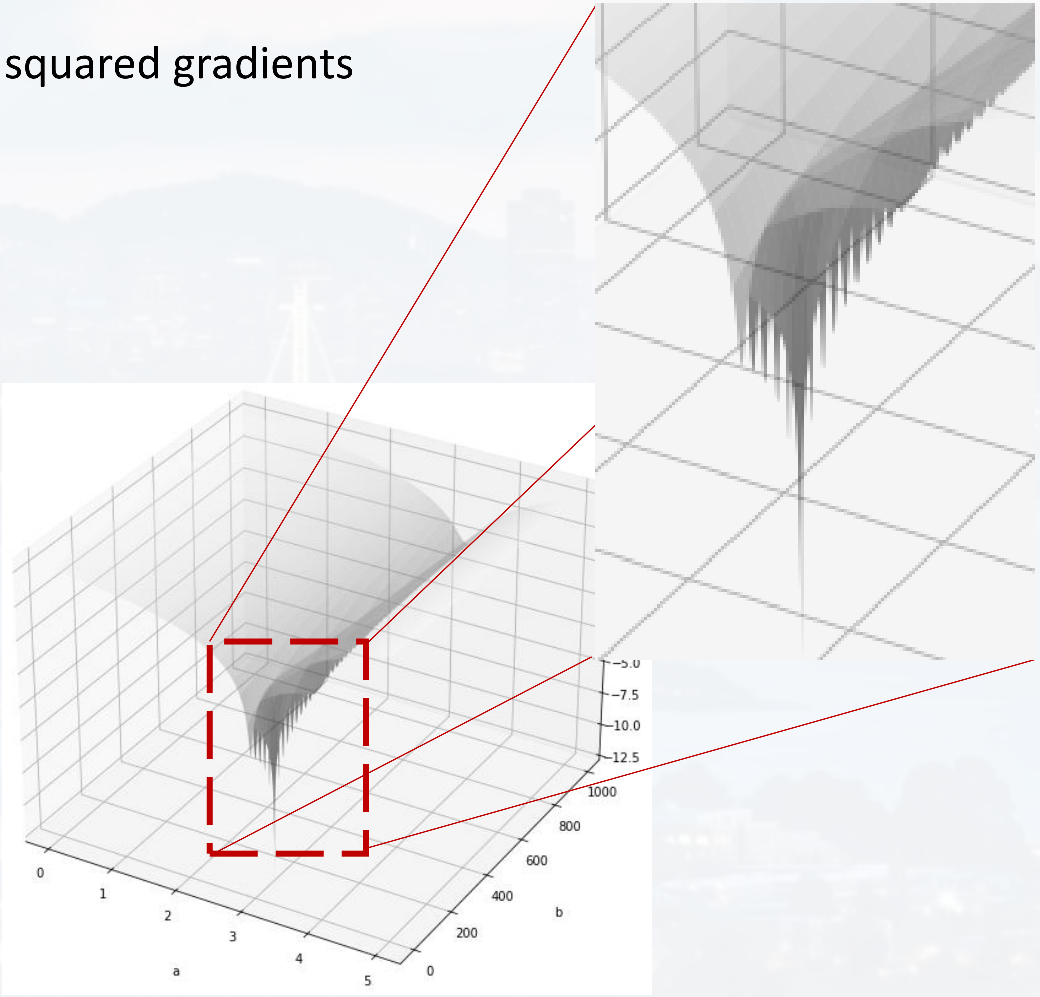
idea: reducing ϵ over time by **accumulated sum** of squared gradients

$$r_{t+1} = r_t + [\nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)]^2$$

$$\epsilon = \frac{\epsilon_{t=0}}{\sqrt{r_{t+1}} + \delta}$$

$\delta > 0$ for stability

problem: can get stuck in a local minimum



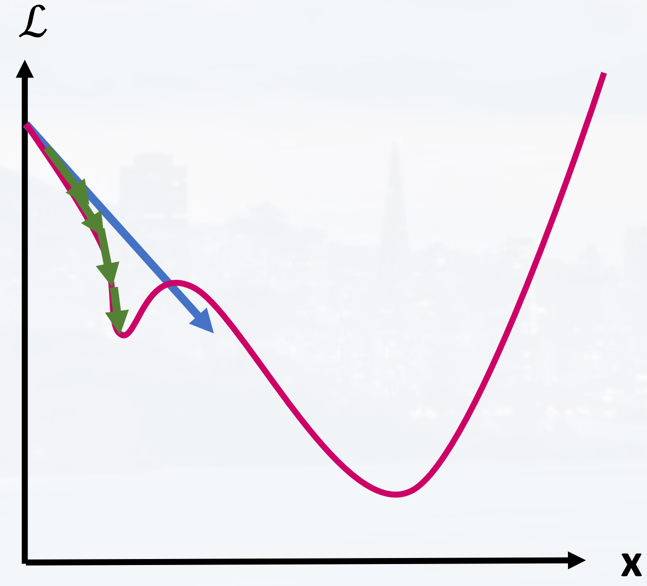
$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)$$

$\epsilon > 0$:	learning rate
t:	iteration
κ:	decay rate
y:	true value
$\hat{y}$:	model prediction
$\mu_{t=0}$:	momentum

momentum:

taking the **average** of ***N*** previous gradients

$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t) \rangle = \frac{1}{N} [\nabla_{\vec{x}} \mathcal{L}(\vec{x}_{t-1}) + \nabla_{\vec{x}} \mathcal{L}(\vec{x}_{t-2}) + \dots + \nabla_{\vec{x}} \mathcal{L}(\vec{x}_{t-N})]$$



but we want more recent gradients to contribute more than older gradients

→ **weighted average** with weighting factor μ_k

$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t) \rangle = \sum_{k=t-N}^{t-1} \mu_k \cdot \nabla_{\vec{x}} \mathcal{L}(\vec{x}_k)$$

find a clever way to adjust μ_k during every iteration *t*

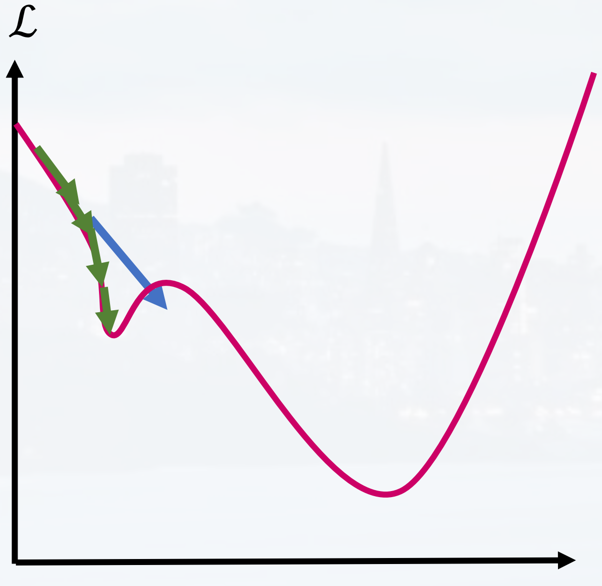
$\epsilon > 0$: learning rate
t: iteration
κ: decay rate
y: true value
$\hat{y}$: model prediction
$\mu_{t=0}$: momentum

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)$$

momentum:

weighted average with weighting factor μ_k

find a clever way to adjust μ_k during every iteration t



$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0) \rangle = \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0) \qquad \mu_0 = (0,1)$$

$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) \rangle = \nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) + \mu_0 \cdot \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0)$$

$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_2) \rangle = \nabla_{\vec{x}} \mathcal{L}(\vec{x}_2) + \mu_0 \cdot [\nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) + \mu_0 \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0)] \qquad \mu_{k=2} = \mu_0 \, \mu_0 = \mu_0^2$$

$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_3) \rangle = \nabla_{\vec{x}} \mathcal{L}(\vec{x}_3) + \boxed{\mu_0} \cdot [\nabla_{\vec{x}} \mathcal{L}(\vec{x}_2) + \boxed{\mu_0} \cdot [\nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) + \boxed{\mu_0} \cdot \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0)]]$$

… and so on…

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \, \nabla_{\vec{x}} \, \mathcal{L}(\vec{x}_t)$$

$\epsilon > 0$:	learning rate
t:	iteration
κ:	decay rate
y:	true value
$\hat{y}$:	model prediction
$\mu_{t=0}$:	momentum

momentum:

weighted average with weighting factor μ_k

find a clever way to adjust μ_k during every iteration t $\mu_0 = (0,1)$

$$\langle \nabla_{\vec{x}} \, \mathcal{L}(\vec{x}_3) \rangle = \nabla_{\vec{x}} \, \mathcal{L}(\vec{x}_3) + \mu_0 \cdot \left[\nabla_{\vec{x}} \, \mathcal{L}(\vec{x}_2) + \mu_0 \cdot \left[\nabla_{\vec{x}} \, \mathcal{L}(\vec{x}_1) + \mu_0 \cdot \nabla_{\vec{x}} \, \mathcal{L}(\vec{x}_0) \right] \right]$$

… and so on…

```python
class Optimizer:

    def __init__(self, learning_rate = 0.1, decay = 0, momentum = 0):
        self.learning_rate         = learning_rate
        self.decay                 = decay
        self.current_learning_rate = learning_rate
        self.iterations            = 0
        self.momentum              = momentum
```

ANN: Perceptron, Backpropagation, SGD

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)$$

$\epsilon > 0$:	learning rate
t:	iteration
λ:	Lagrangian Multiplier
y:	true value
$\hat{y}$:	model prediction
$\mu_{t=0}$:	momentum

regularization: recall: linear regression

$$\hat{\beta} = \underset{\beta}{argmin} \left\{ \frac{1}{N} \|Y - X\beta\|^2 + \boldsymbol{\lambda\|\beta\|^1} \right\}$$

the Loss Function
$\boldsymbol{L}(\boldsymbol{X}, \boldsymbol{Y}, \lambda)$

L1 or **L**east **a**bsolute **s**hrinkage and **s**election **o**perator
- encourages **sparsity** of β
- reduces **overfitting**

$$\hat{\beta} = \underset{\beta}{argmin} \left\{ \frac{1}{N} \|Y - X\beta\|^2 + \boldsymbol{\lambda\|\beta\|^2} \right\}$$

L2 or **Ridge**
- **penalizes large $\boldsymbol{\beta}$**

$$\hat{\beta} = \underset{\beta}{argmin} \left\{ \frac{1}{N} \|Y - X\beta\|^2 + \boldsymbol{\lambda\, max(0, -\beta)} \right\}$$

- **penalizes negative $\boldsymbol{\beta}$**

...and so on

ANN: Perceptron, Backpropagation, SGD

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)$$

$\epsilon > 0$:	learning rate
t:	iteration
λ:	Lagrangian Multiplier
y:	true value
$\hat{y}$:	model prediction
$\mu_{t=0}$:	momentum

regularization:

L1 and L2 regularization

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}}[\mathcal{L}(\vec{x}_t) + \lambda_1 ||\vec{x}_t||^1 + \lambda_2 ||\vec{x}_t||^2]$$

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t) \; - \epsilon\lambda_1 \nabla_{\vec{x}} ||\vec{x}_t||^1 \; - \epsilon\lambda_2 \nabla_{\vec{x}} ||\vec{x}_t||^2$$

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)$$

$\epsilon > 0$:	learning rate
t:	iteration
λ:	Lagrangian Multiplier
y:	true value
$\hat{y}$:	model prediction
$\mu_{t=0}$:	momentum

regularization:

L1 and L2 regularization

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}}[\mathcal{L}(\vec{x}_t) + \lambda_1 ||\vec{x}_t||^1 + \lambda_2 ||\vec{x}_t||^2]$$

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t) \; - \epsilon\lambda_1 \nabla_{\vec{x}} ||\vec{x}_t||^1 \; - \epsilon\lambda_2 \nabla_{\vec{x}} ||\vec{x}_t||^2$$

notes:

-vanilla gradient descent does not stop if values for $\vec{x}$ are too large

-the derivative of $||x||^1$ returns the sign (i. e. direction) and therefore encourages sparsity (**reduces overfitting**)

-usually $\lambda \ll ||\vec{x}_t||^n$

-will be important for ANNs later

ANN: Perceptron, Backpropagation, SGD

$$\vec{x}_{t+1} = \vec{x}_t - \epsilon \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)$$

$\epsilon > 0$:	learning rate
t:	iteration
λ:	Lagrangian Multiplier
y:	true value
$\hat{y}$:	model prediction
$\mu_{t=0}$:	momentum

Root Mean Square Propagation:

problem: Adagrad accumulates all previous gradients → slows down updates too fast

$$r_{t+1} = r_t + [\nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)]^2 \qquad \epsilon = \frac{\epsilon_{t=0}}{\sqrt{r_{t+1}} + \delta} \qquad \delta > 0 \text{ for stability}$$

idea: weighted average like for **momentum**

$$r_{t+1} = \beta\, r_t + (1-\beta)[\nabla_{\vec{x}} \mathcal{L}(\vec{x}_t)]^2 \qquad \beta = (0,1)$$

$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) \rangle = \nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) + \mu_0 \cdot \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0)$$ **momentum**

$$\epsilon \rightarrow \frac{\epsilon}{\sqrt{r_{t+1}} + \delta}$$ adaptive gradient, aka **AdaGrad**

Root Mean Square Propagation **RMSProp**

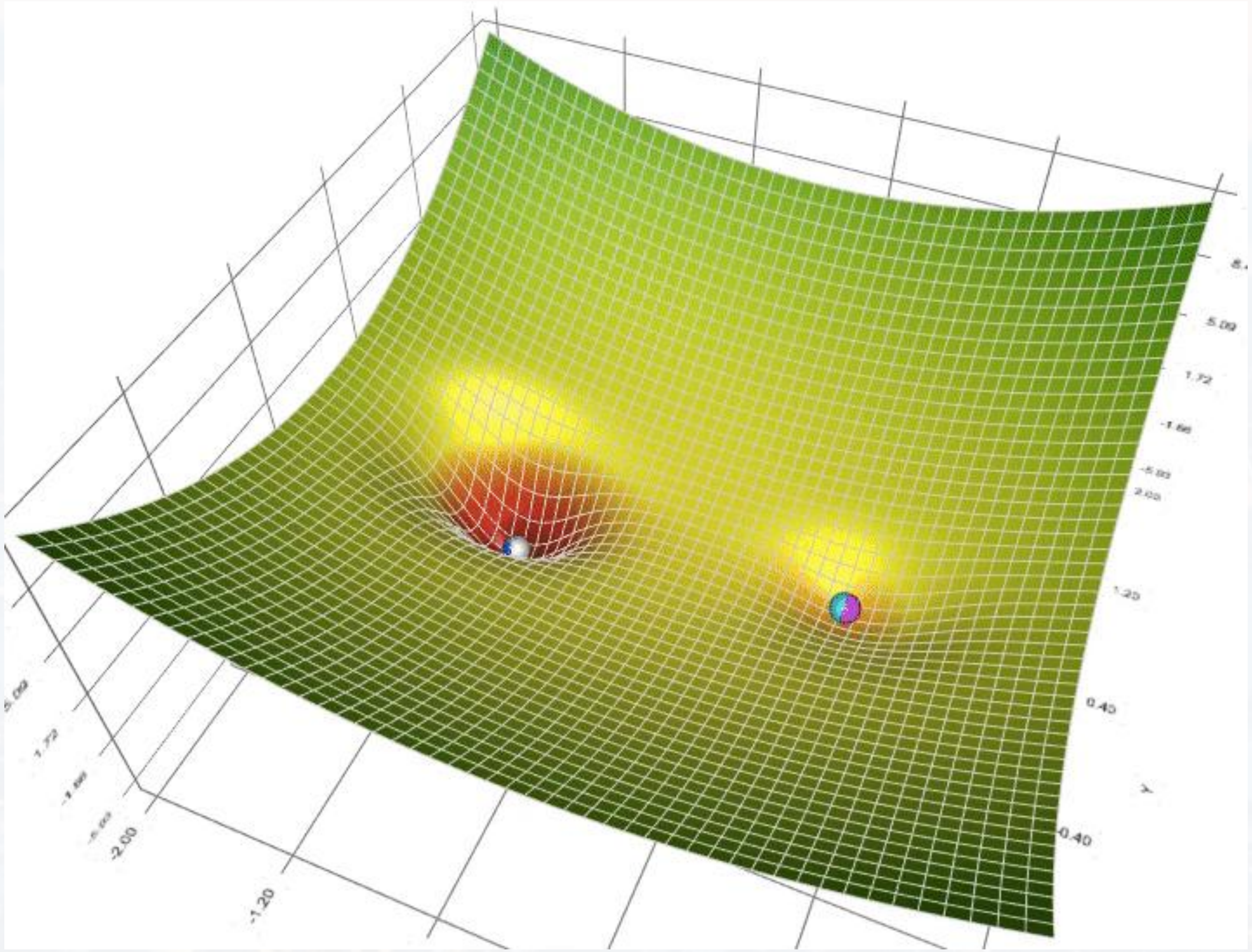
all combined:
Adaptive Moment Estimation
aka **Adam**

ANN: Perceptron, Backpropagation, SGD

Gradient Descent

Lili Jiang

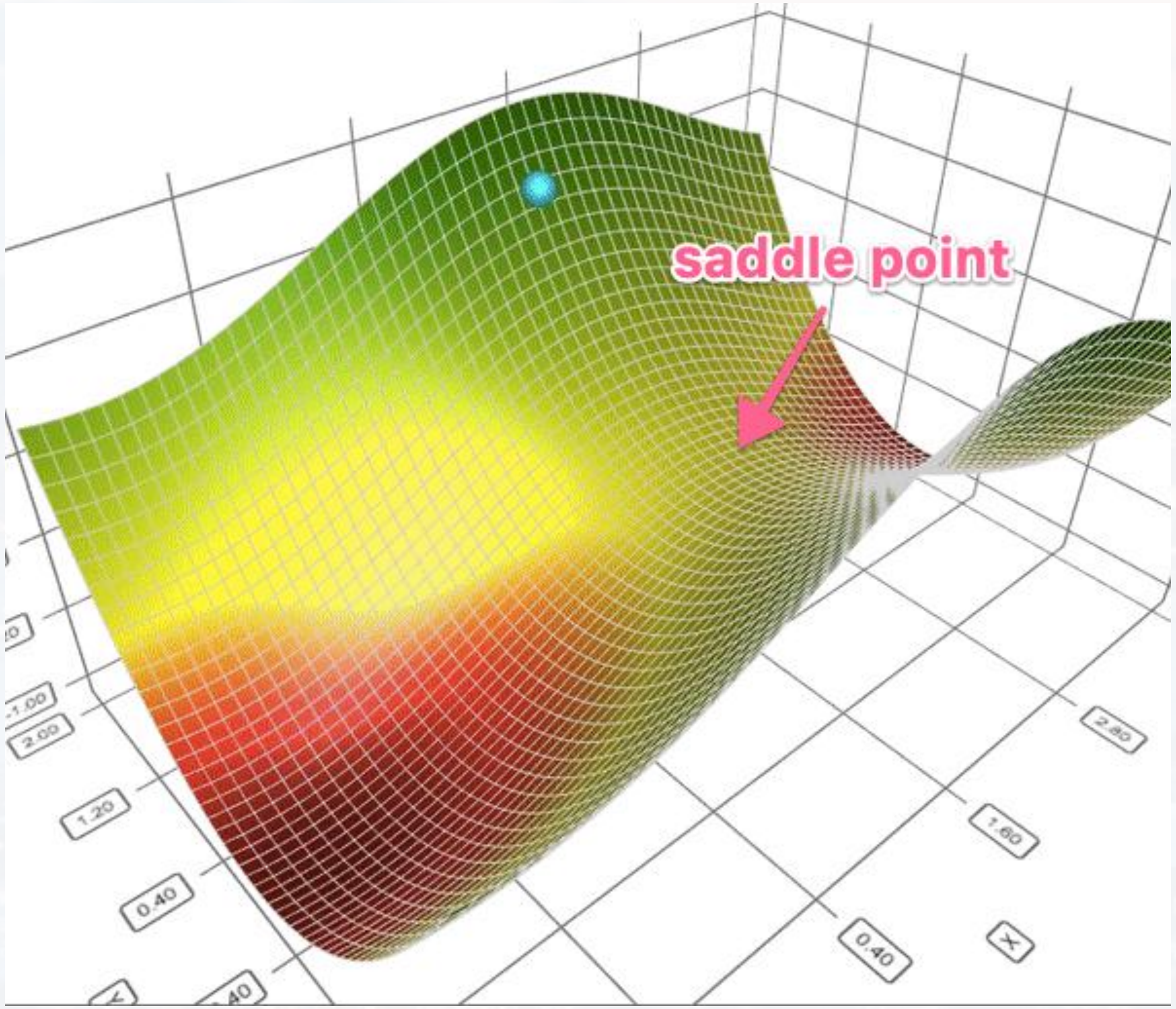
TowardsDataScience



gradient descent (cyan),
momentum (magenta),
AdaGrad (white),
RMSProp (green),
Adam (blue)

Lili Jiang

[TowardsDataScience]



gradient descent (cyan),
momentum (magenta),
AdaGrad (white),
RMSProp (green),
Adam (blue)

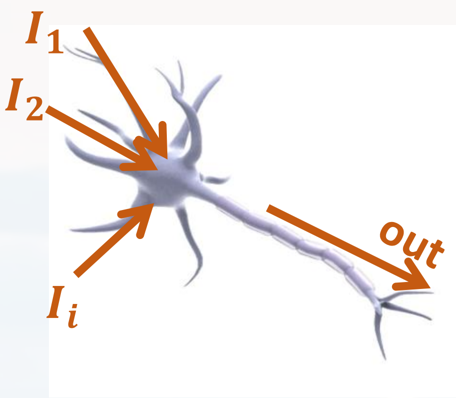
see also
`WalkThroughGradDescent.ipynb`

Outline

- Gradient Descent
  - Vanilla
  - Learning Rate Schedule
  - L1 and L2
  - Momentum
  - AdaGrad
  - RMSProp
- **Perceptron**
- Backpropagation

ANN: Perceptron, Backpropagation, SGD: Perceptron

$\boldsymbol{b}$:	bias (base potential
$\boldsymbol{I_i}$:	input *i*
$\boldsymbol{w_i}$:	corresponding weight



$$net = \sum_i I_i \cdot w_i + b$$

weights and bias are **"learnable"**

recall: linear models

$$y_k = \beta_0 + \sum_{n=1}^{N} \beta_n x_n + \epsilon$$

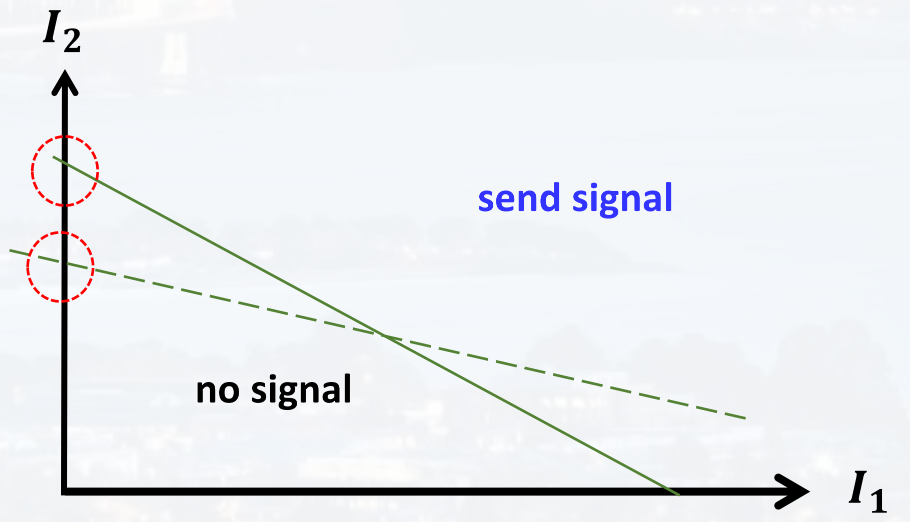
$$\begin{pmatrix} y_1 \\ \dots \\ y_k \\ \dots \\ \dots \\ y_K \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & x_{12} \; \dots \; x_{1n} \; \dots \; x_{1N} \\ \dots & \dots & \dots \\ 1 & x_{k1} & x_{kn} \\ 1 & \dots & \dots \\ 1 & x_{K1} & x_{K2} \; \dots \; x_{Kn} \; \dots \; x_{KN} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \dots \\ \beta_n \\ \dots \\ \beta_N \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \dots \\ \varepsilon_k \\ \dots \\ \varepsilon_K \end{pmatrix}$$

<u>simple example:</u>

one neuron with a **switch**, threshold **T** and **two** input channels

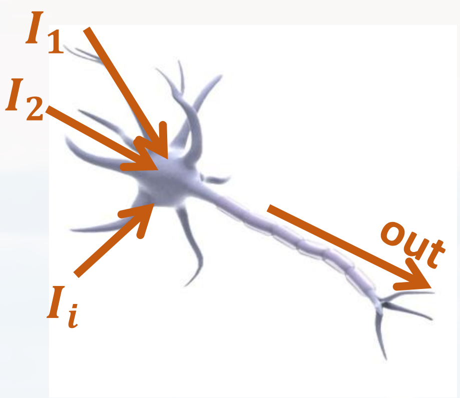
fire if: $b + I_1 w_1 + I_2 w_2 > T$

$$I_2 = -\frac{w_1}{w_2} I_1 + \frac{T - b}{w_2}$$



ANN: Perceptron, Backpropagation, SGD:

b:	bias (base potential
I_i:	input *i*
w_i:	corresponding weight



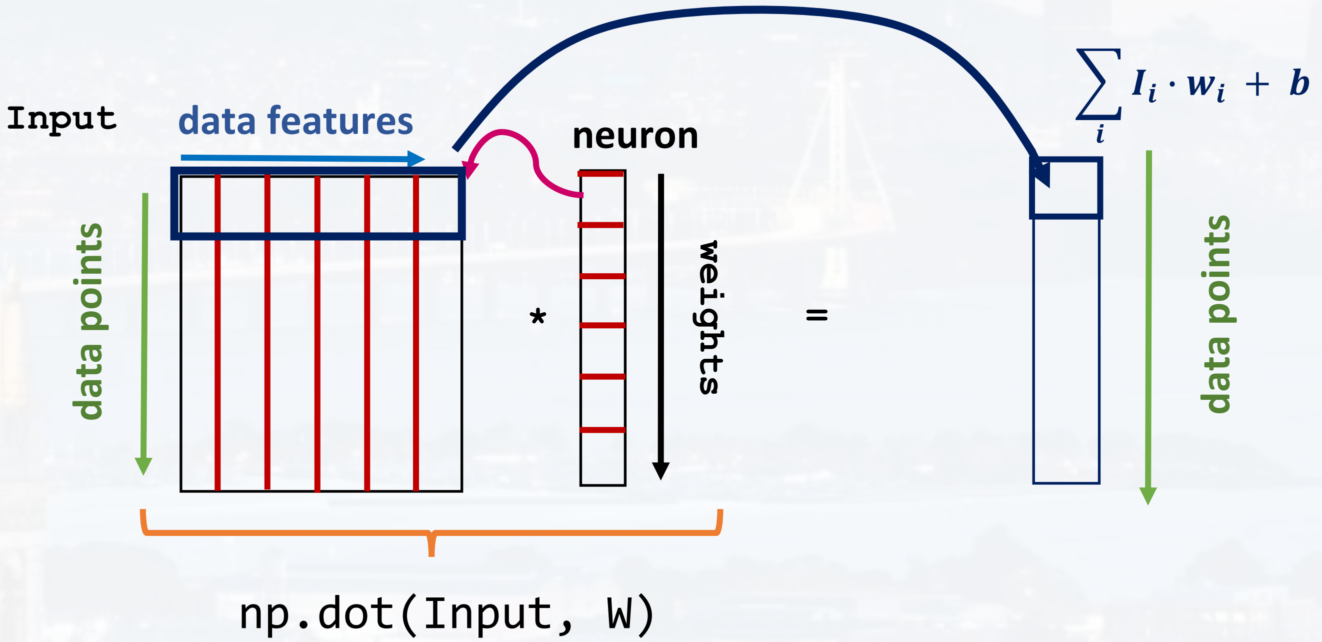
$$net = \sum_i I_i \cdot w_i + b$$

weights and bias are **"learnable"**

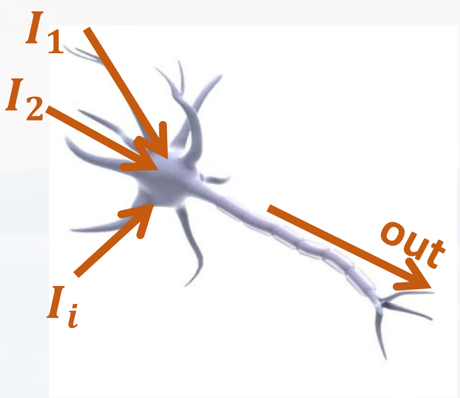
data features

data points

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
```



ANN: Perceptron, Backpropagation, SGD:



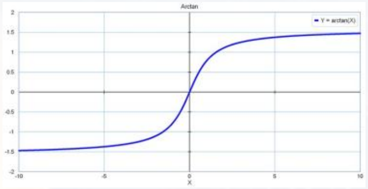
$$net = \sum_i I_i \cdot w_i + b$$

weights and bias are **"learnable"**

$\boldsymbol{b}$:	bias (base potential
$\boldsymbol{I_i}$:	input *i*
$\boldsymbol{w_i}$:	corresponding weight

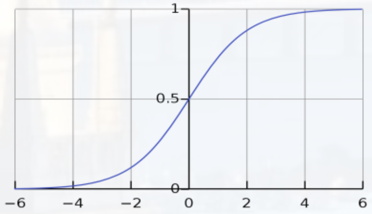
net → some activation function $\boldsymbol{f}$ → actual output

- y = arctan(net)



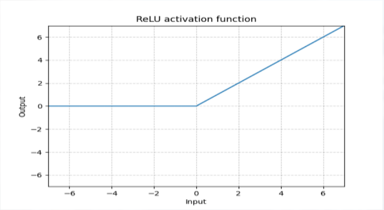
$(-\infty;\ +\infty) \rightarrow (-\pi/2;\ +\pi/2)$

- y = sigm(net)



$(-\infty;\ +\infty) \rightarrow (0; 1)$

- y = ReLU(net) = *max(0, net)*



$(-\infty;\ +\infty) \rightarrow (0; +\infty)$

…and many other

Outline

- **Gradient Descent**
  - Vanilla
  - Learning Rate Schedule
  - L1 and L2
  - Momentum
  - AdaGrad
  - RMSProp
- **Perceptron**
- **Backpropagation**

$\boldsymbol{b}$:	**bias (base potential**
$\boldsymbol{I_i}$:	**input *i***
$\boldsymbol{w_i}$:	**corresponding weight**

learning:

$$net = \sum_i I_i \cdot w_i + b \xrightarrow{\text{activation function } \boldsymbol{f}} y = f(net) \xrightarrow{\text{loss function } \mathcal{L}} \mathcal{L} = \frac{1}{2}(t - y)^2$$

target output ***t***

finding best w_i by **minimizing** $\mathcal{L}$

$$\Delta w_i = w_i(new) - w_i(old) = -\epsilon \frac{\partial \mathcal{L}}{\partial w_i}$$

gradient descent with learning rate ϵ

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial net}\frac{\partial net}{\partial w_i} = \boxed{-(t-y)\ f'(net)}\ \boxed{I_i}$$

outer derivatives — inner derivative

The required change of $\mathcal{L}$ propagates back to the changes of $\boldsymbol{w_i}$ ***and b*** → **backpropagation**

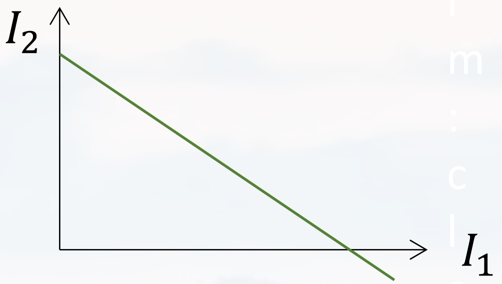
$$\boxed{\Delta w_i = w_i(new) - w_i(old) = \epsilon(t-y)\ f'(net)\ I_i}$$

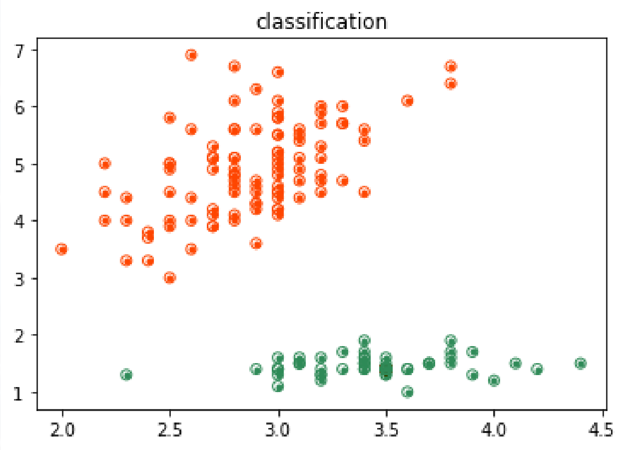
$$\boxed{\Delta b = b(new) - b(old) = \epsilon(t-y)\ f'(net)\ \cdot 1}$$

see `Perceptron.ipynb`

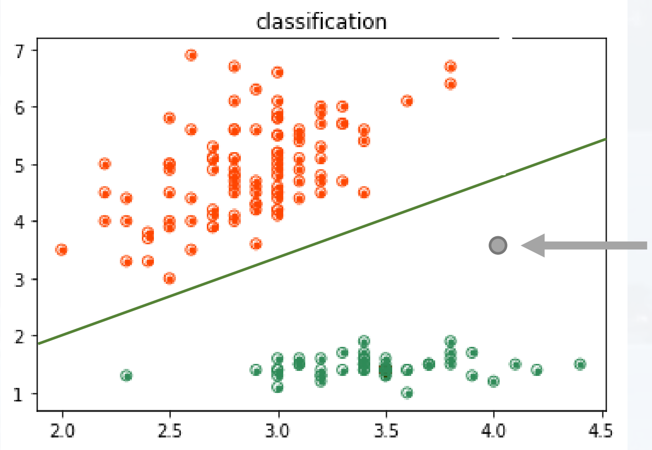
see `Perceptron.ipynb`

one neuron
two features
two classes
N data points





goal

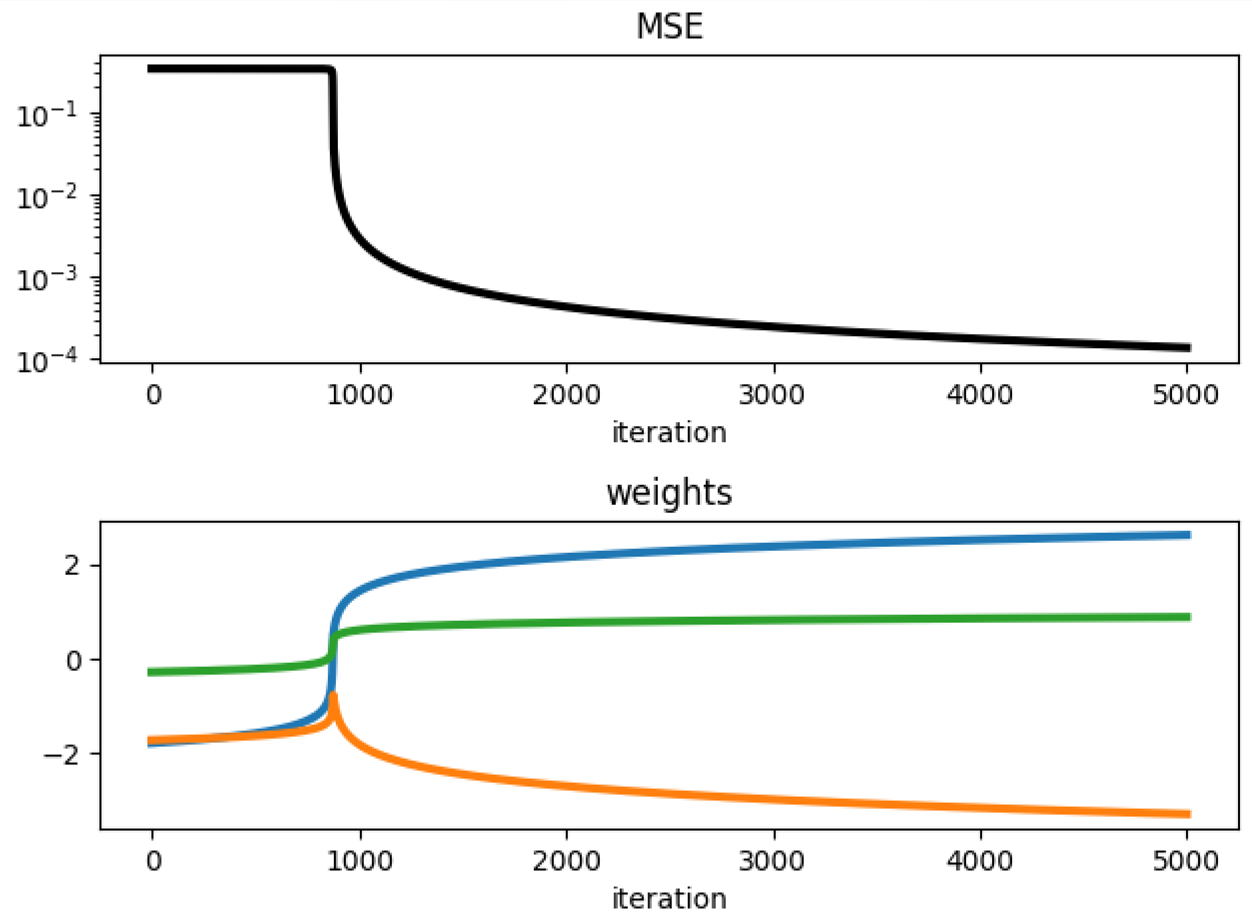


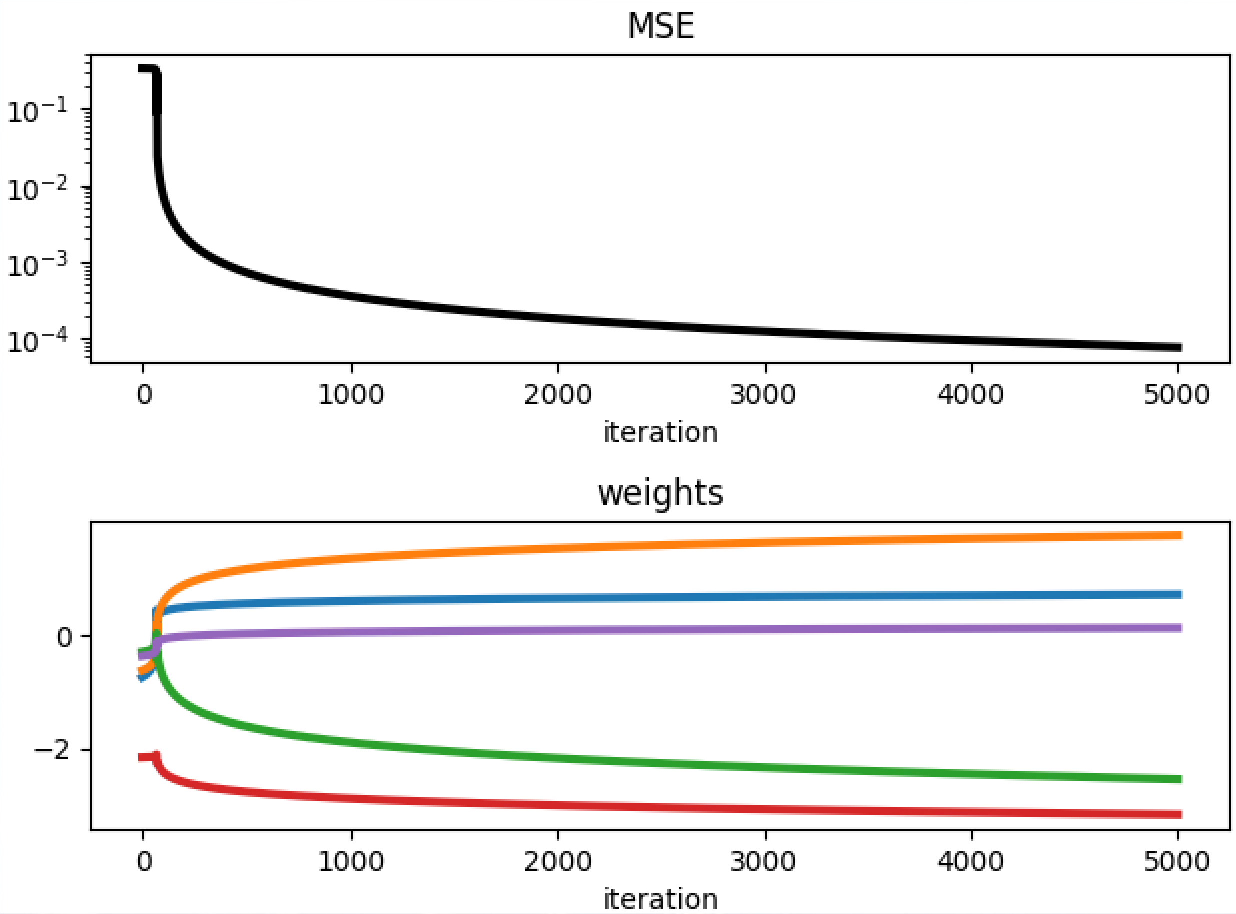
classifying a new data point

ANN: Perceptron, Backpropagation, SGD

see `Perceptron.ipynb`

the training process

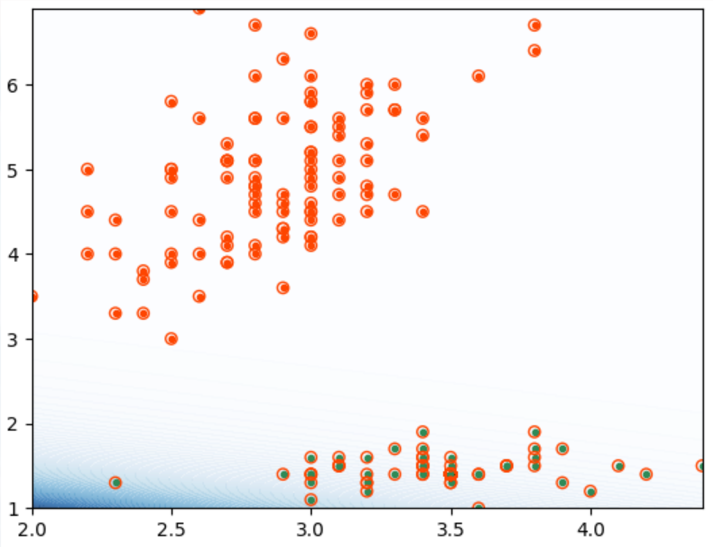




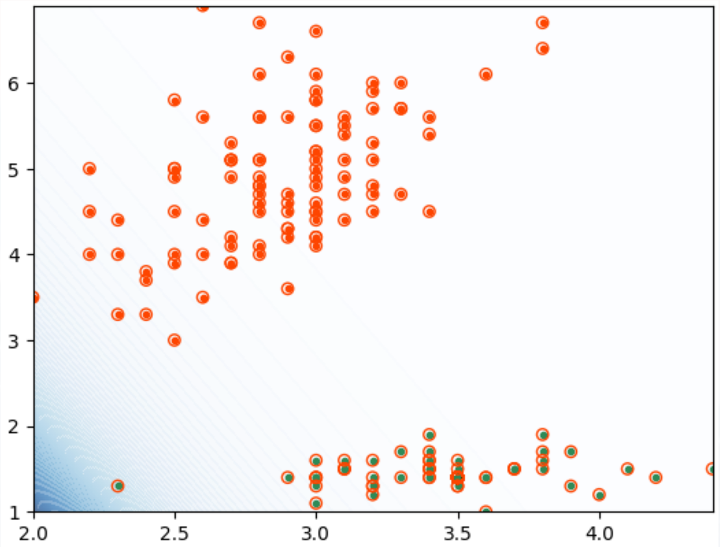
ANN: Perceptron, Backpropagation, SGD

see `Perceptron.ipynb`

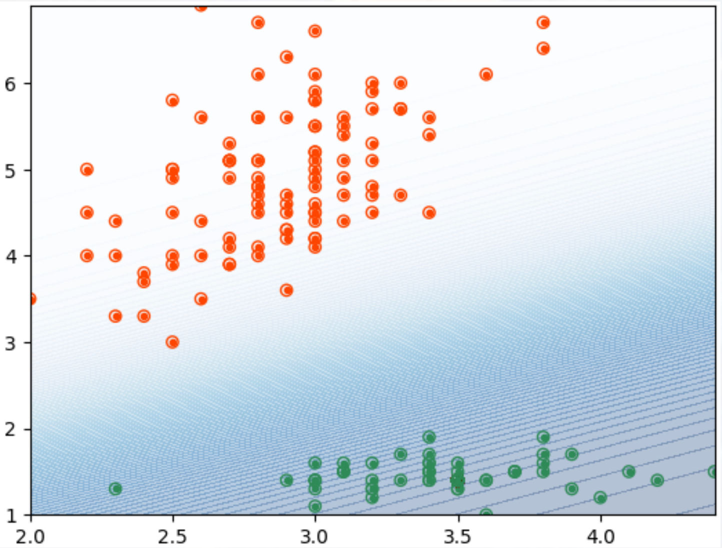
the training process



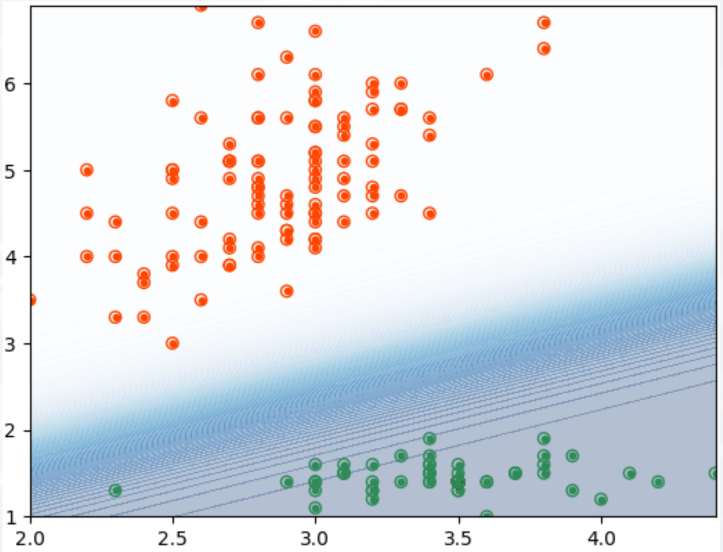
t = 3



t = 5



t = 100



t = 10,000

ANN: Perceptron, Backpropagation, SGD

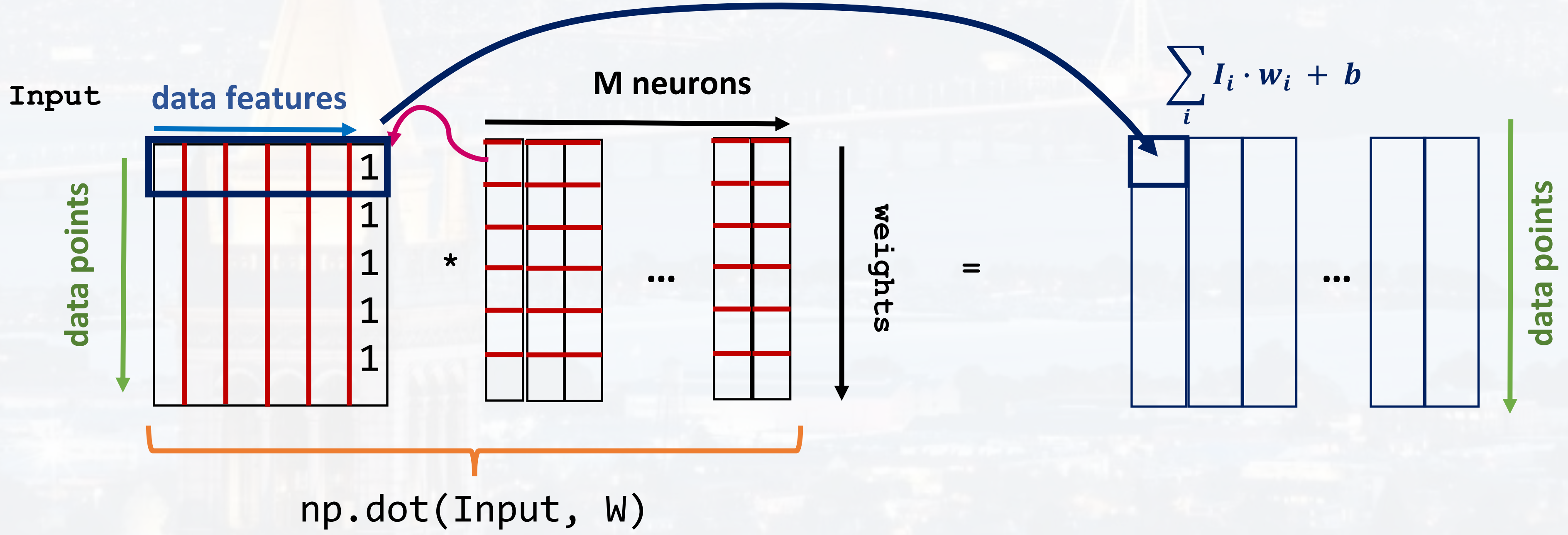
b:	bias (base potential
I_i:	input *i*
w_i:	corresponding weight

dense layer: **multiple neurons**

data features

data points

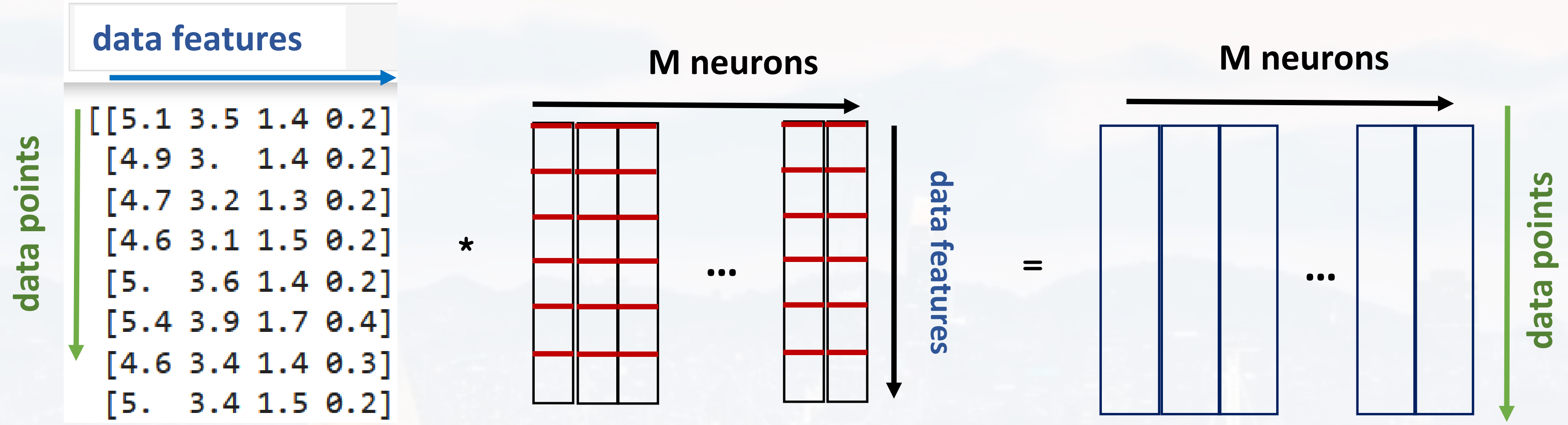
```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
```



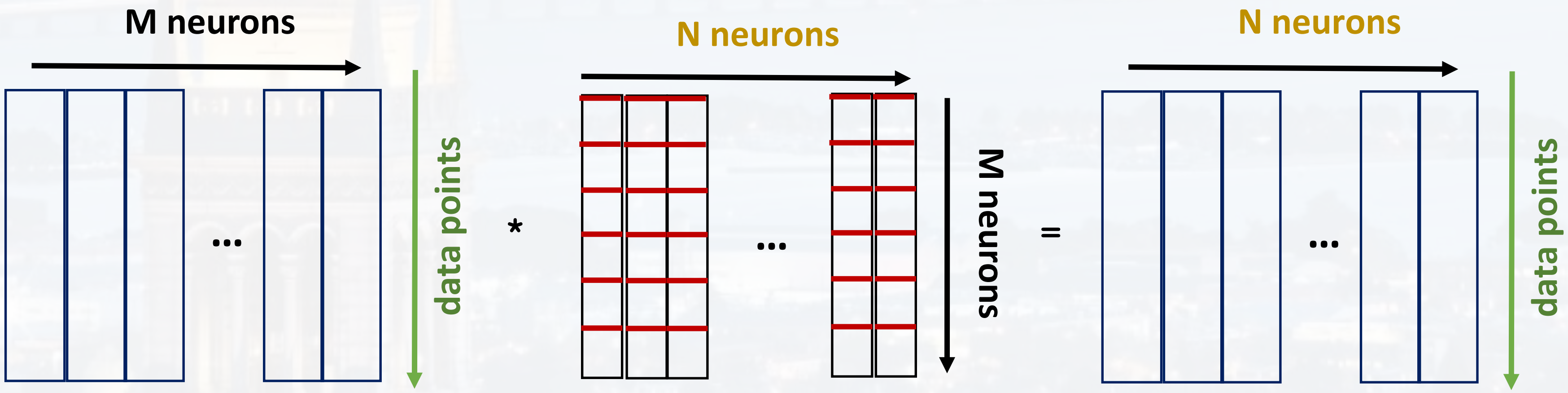
ANN: Perceptron, Backpropagation, SGD

adding subsequent dense layer

b:	bias (base potential
I_i:	input *i*
w_i:	corresponding weight



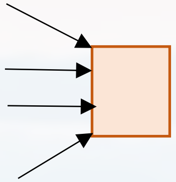
input layer



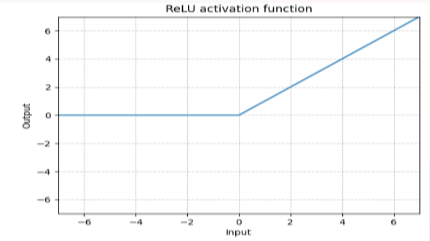
hidden layer

ANN: Perceptron, Backpropagation, SGD

for training: building the backpropagation part



$\sum_i I_i w_i + b = net$



$y = f(net)$

… loss

$L(w_i, b, I_i)$

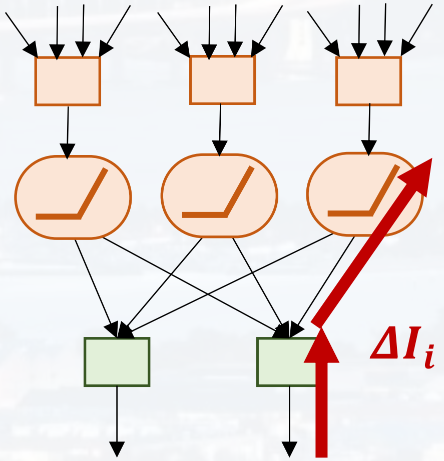
e. g.: - MSE
- accuracy
- etc

$\boxed{dy}$ $\boxed{dL}$

$$\Delta w_i = -\epsilon \boxed{\frac{dL}{dy}} \boxed{\frac{dy}{dnet}} I_i$$

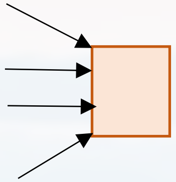
$$\Delta I_i = -\epsilon \boxed{\frac{dL}{dy}} \boxed{\frac{dy}{dnet}} w_i \quad \text{from} \quad \frac{\partial L}{\partial I_i}$$

$$\Delta b = -\epsilon \boxed{\frac{dL}{dy}} \boxed{\frac{dy}{dnet}} 1 \quad \text{from} \quad \frac{\partial L}{\partial b}$$

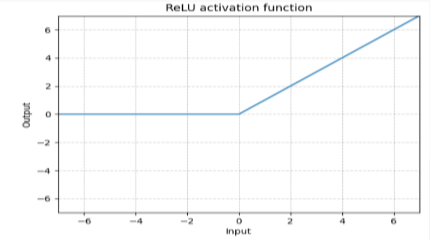


ANN: Perceptron, Backpropagation, SGD

for training: building the backpropagation part



$$\sum_i I_i w_i + b = net$$



$$y = f(net)$$

→ … → loss

$L(w_i, b, I_i)$

e. g.:
- MSE
- accuracy
- etc

$\boxed{dy}$ $\boxed{dL}$

$$\Delta w_i = -\epsilon \boxed{\frac{dL}{dy}} \boxed{\frac{dy}{dnet}} I_i$$

$$\Delta I_i = -\epsilon \boxed{\frac{dL}{dy}} \boxed{\frac{dy}{dnet}} w_i \quad \text{from} \quad \frac{\partial L}{\partial I_i}$$

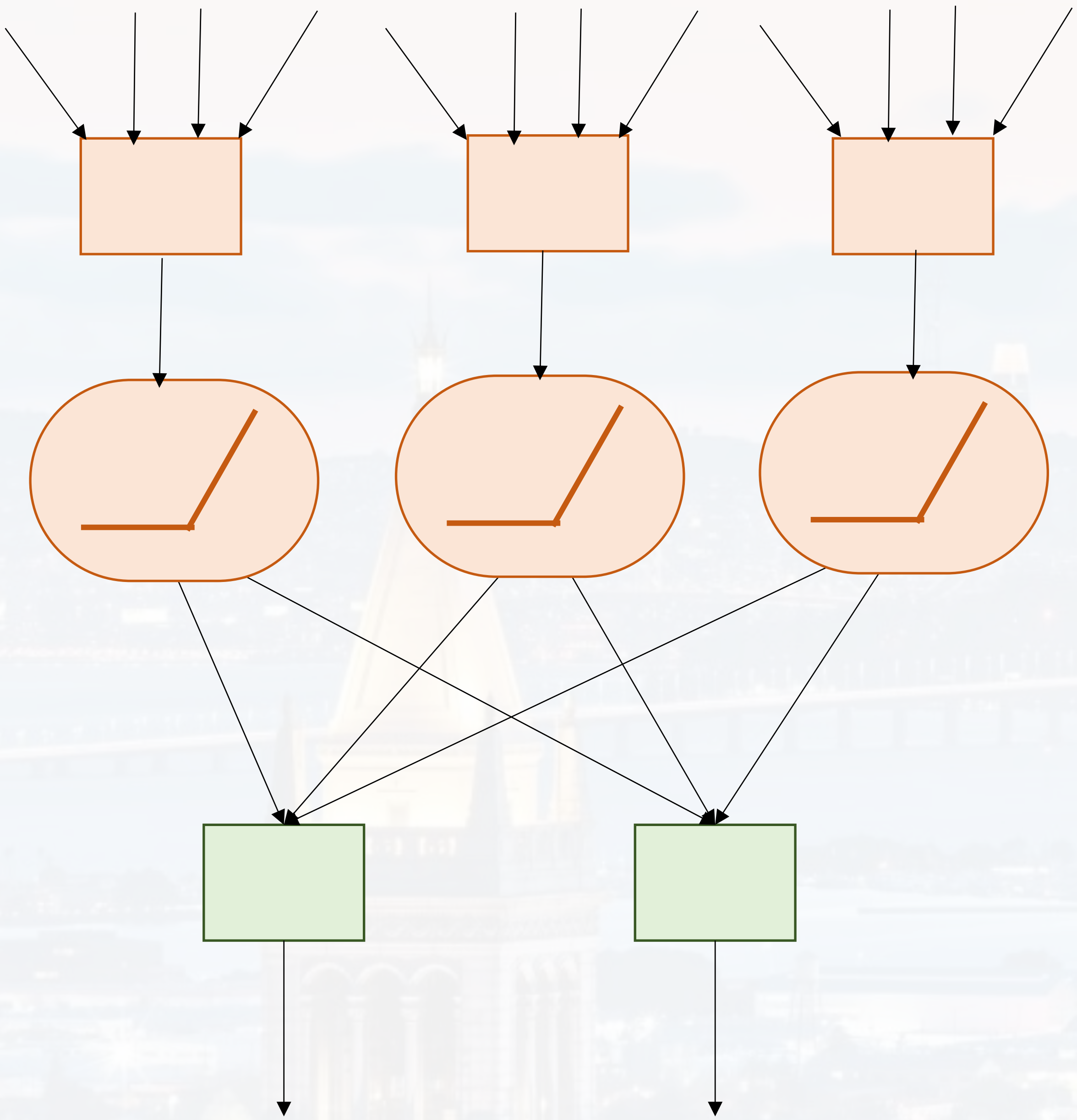
$$\Delta b = -\epsilon \boxed{\frac{dL}{dy}} \boxed{\frac{dy}{dnet}} 1 \quad \text{from} \quad \frac{\partial L}{\partial b}$$

hence, we need to include the following structure for backpropagation:

```
self.dweights  = inputs  * dvalues
self.dinputs   = weights * dvalues
self.dbiases   = 1       * dvalues
```

inner derivative	**product of outer derivatives**

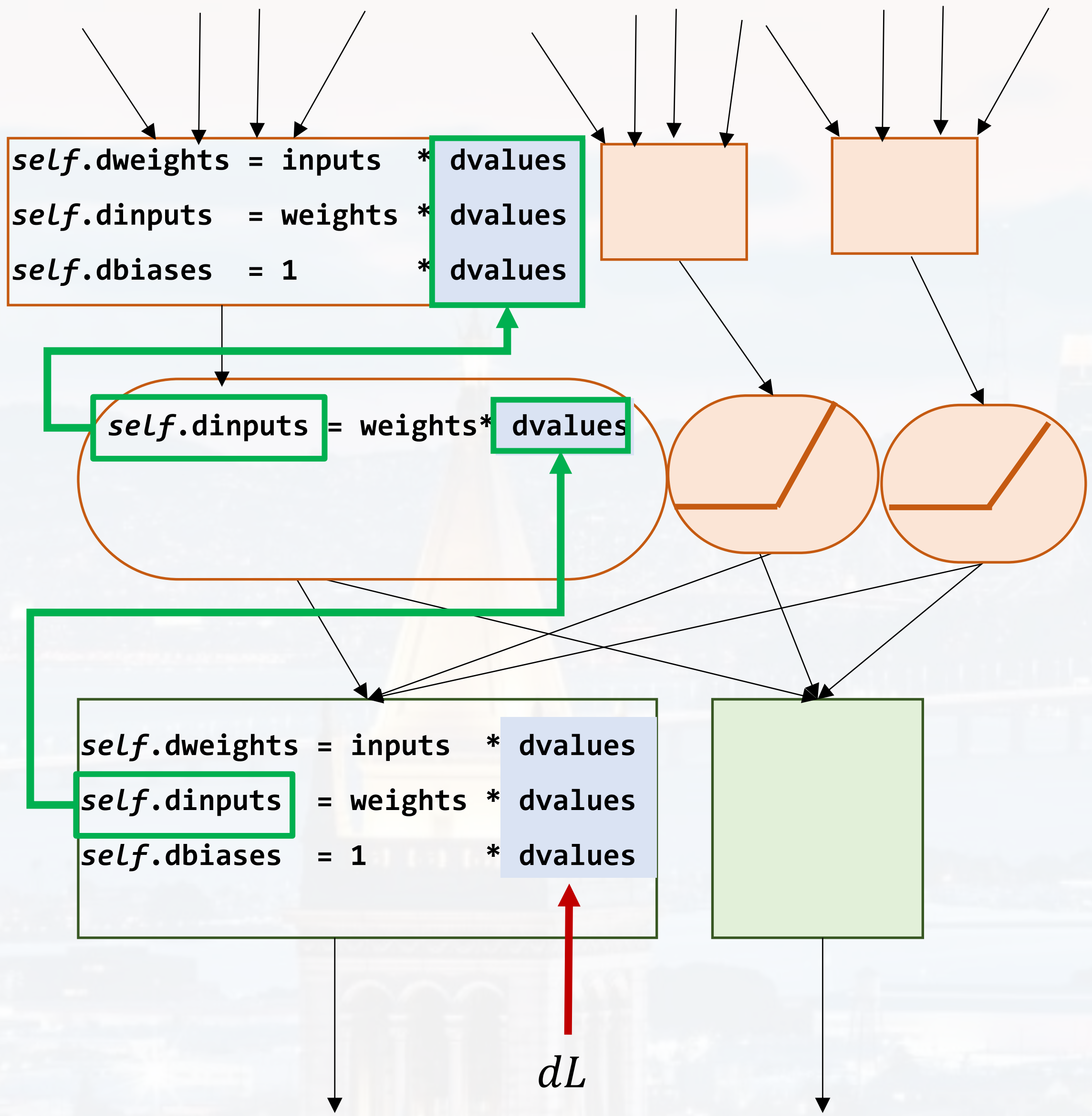
ANN: Perceptron, Backpropagation, SGD



hence, we need to include the following structure for backpropagation:

self.dweights	=	inputs	*	dvalues
self.dinputs	=	weights	*	dvalues
self.dbiases	=	1	*	dvalues
		inner derivative		**product of outer derivatives**

ANN: Perceptron, Backpropagation, SGD



hence, we need to include the following structure for backpropagation:

```
self.dweights  = inputs  * dvalues
self.dinputs   = weights * dvalues
self.dbiases   = 1       * dvalues
```

inner derivative

product of outer derivatives

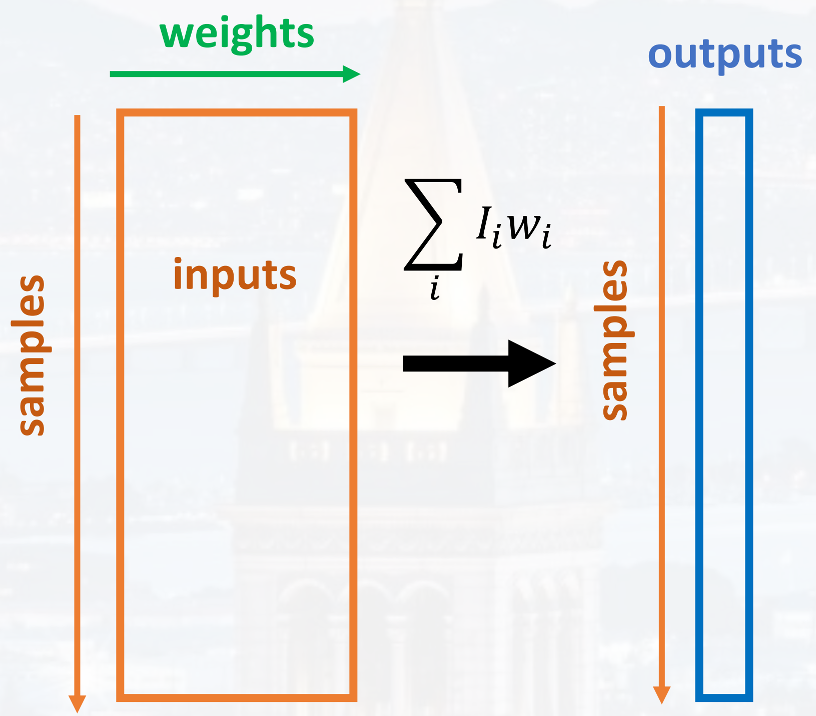
$$\Delta I_i = -\epsilon \boxed{\frac{dL}{dy}} \boxed{\frac{dy}{dnet}} w_i$$

ANN: Perceptron, Backpropagation, SGD

We need **a forward** part and a **backward** part!

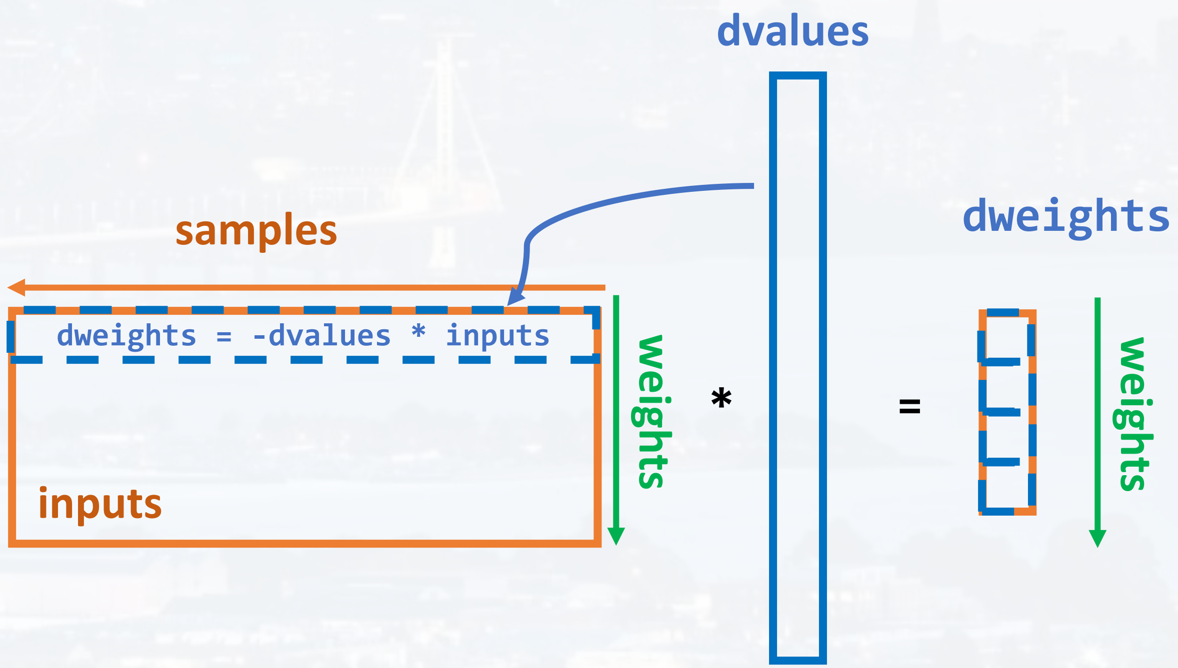
<u>single neuron:</u>

`forward`



`backward`

`dweights = -dvalues * inputs`



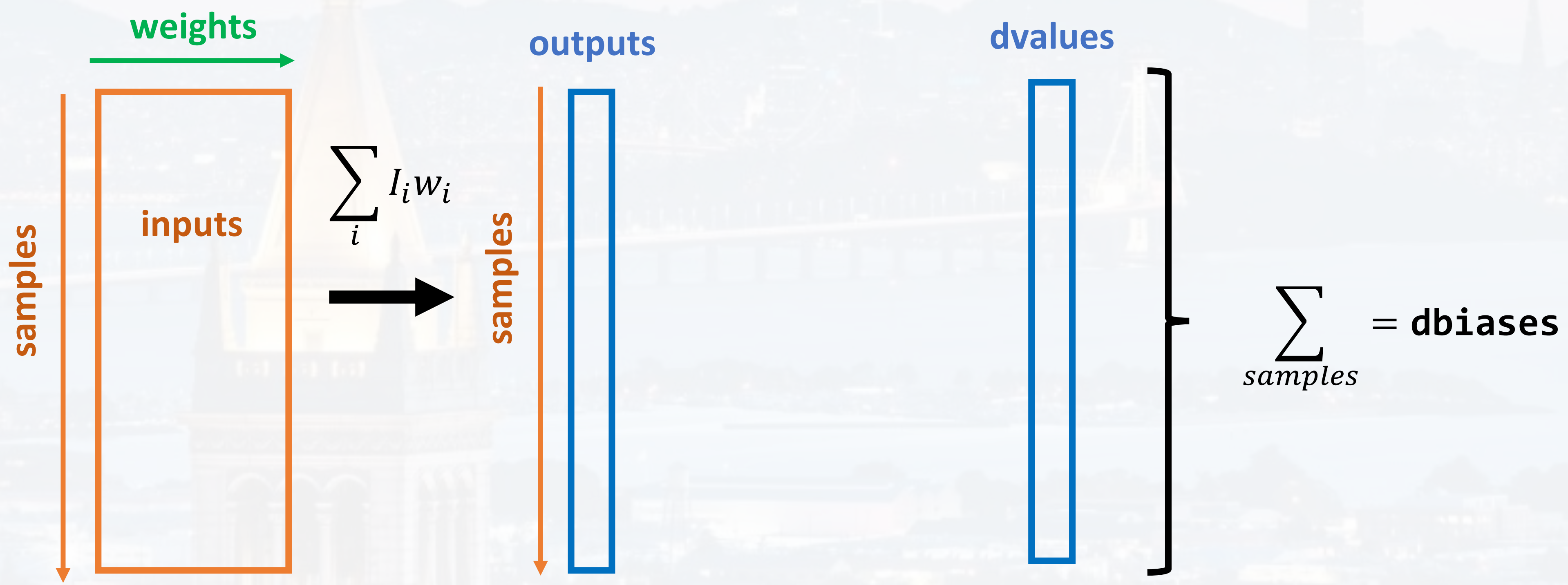
We need **a forward** part and a **backward** part!

<u>single neuron:</u>

`forward`

`backward`

`dbiases = -dvalues`



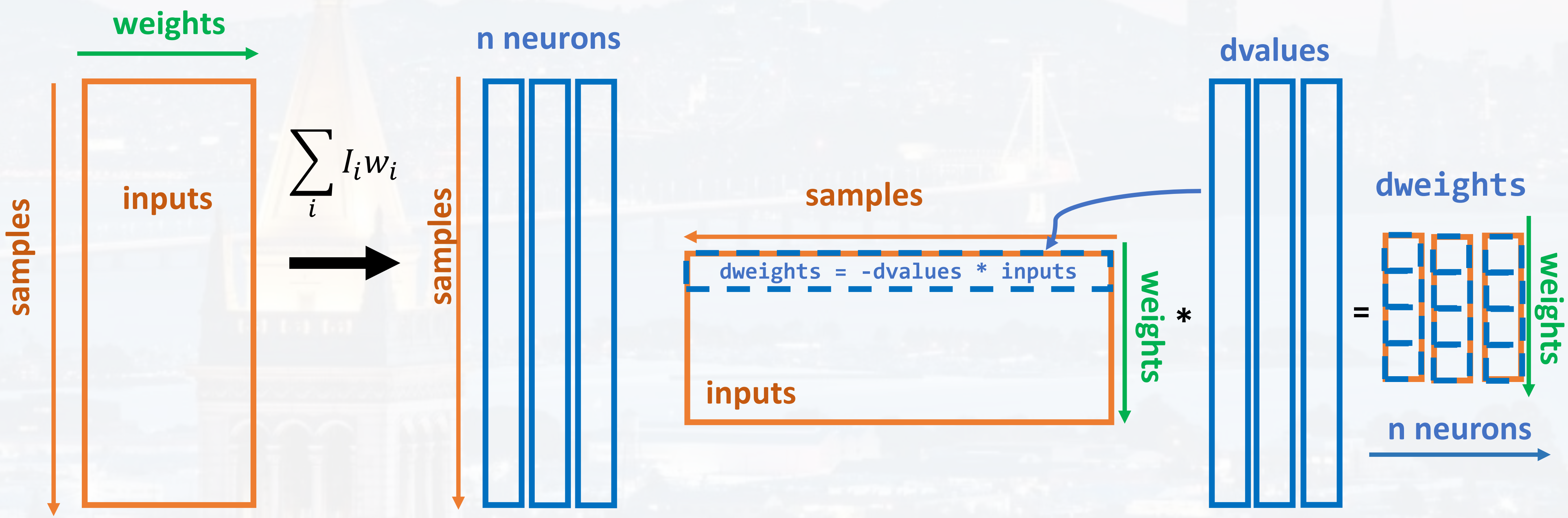
We need **a forward** part and a **backward** part!

dense layer (say **three** neurons):

`forward`

`backward`

`dweights = -dvalues * inputs`



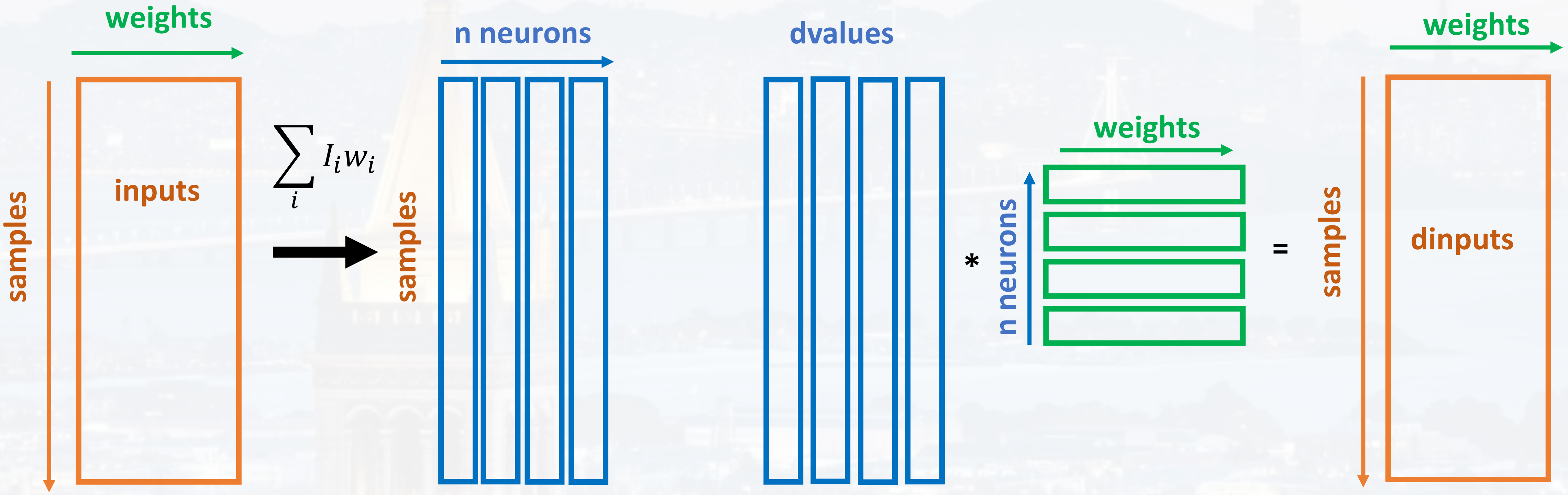
We need **a forward** part and a **backward** part!

dense layer (say **four** neurons):

`forward`

`backward`

`dinputs = -dvalues * weights` **(not needed for first layer!)**



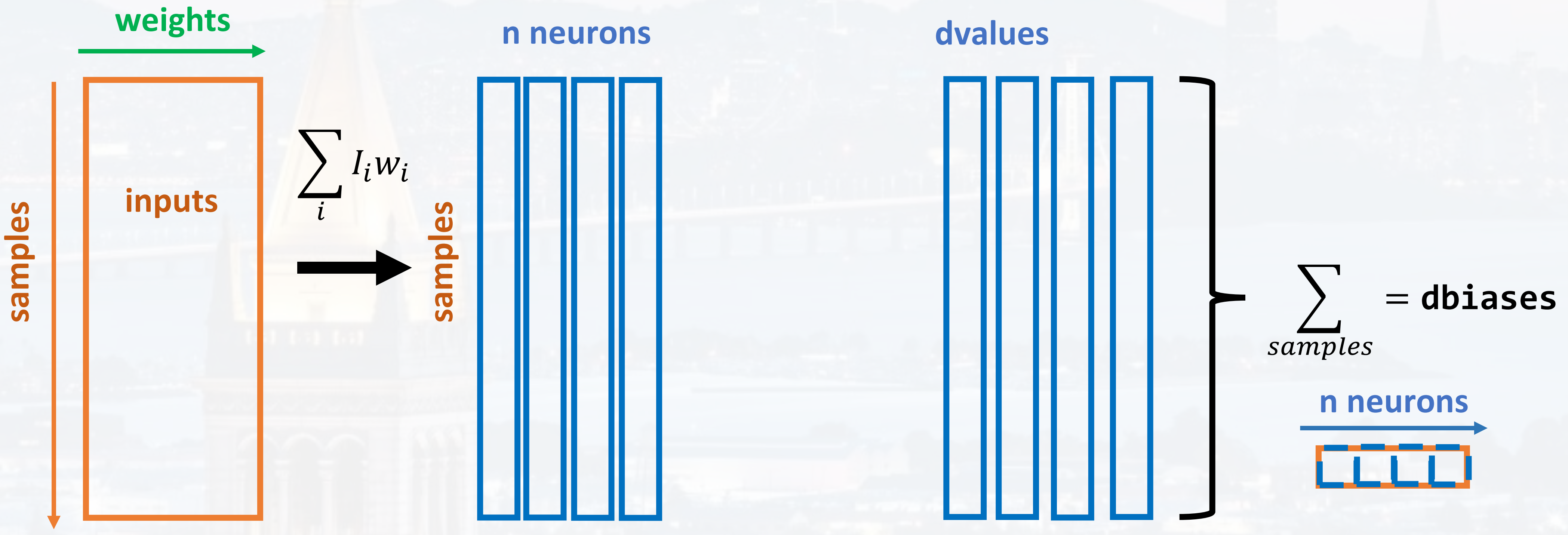
We need **a forward** part and a **backward** part!

dense layer (say **four** neurons):

forward

backward

```
dbiases = -dvalues
```



ANN: Perceptron, Backpropagation, SGD

number of features

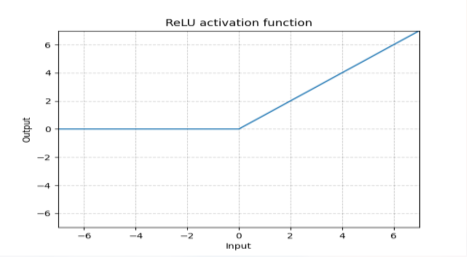
outer derivative

```python
class Layer_Dense():

    def __init__(self, n_inputs, n_neurons):
        self.weights = np.random.randn(n_inputs, n_neurons)
        self.biases  = np.zeros((n_neurons,))


    def forward(self, inputs):
        self.output  = np.dot(inputs, self.weights) + self.biases
        self.inputs  = inputs


    def backward(self, dvalues):

        self.dweights  = np.dot(self.inputs.T, dvalues)
        self.dinputs   = np.dot(dvalues, self.weights.T)
        self.dbiases   = np.sum(dvalues, axis = 0, keepdims = True)
```

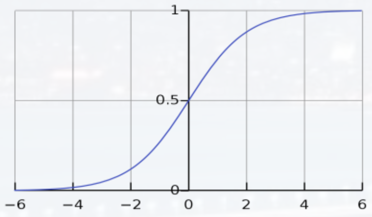
```
self.dweights      = inputs  * dvalues
self.dinputs       = weights * dvalues
self.dbiases       = 1       * dvalues
```

inner derivative **product of outer derivatives**

ANN: Perceptron, Backpropagation, SGD




```python
class Activation_ReLU():

        def forward(self, inputs):
                self.output  = np.maximum(0, inputs)
                self.inputs  = inputs

        def backward(self, dvalues):
                self.dinputs = dvalues.copy()
                self.dinputs[self.inputs <= 0] = 0 #ReLU derivative
```




```python
class Activation_Sigmoid():

        def forward(self, inputs):
                self.output = np.clip(1/(1 + np.exp(-inputs)), 1e-7, 1-1e-7)
                self.inputs = inputs

        def backward(self, dvalues):
                sigm         = self.output
                deriv        = np.multiply(sigm, (1 - sigm))
                self.dinputs = np.multiply(deriv, dvalues)
```

basic structure:

```
alpha = 0.001 #learning rate
```

forward

```
dense1.forward(X)
ReLU.forward(dense1.output)
dense_reg.forward(ReLU.output)
```

evaluation

```
Ypred = dense_reg.output
dE    = Ypred - Target
MSE   = np.sum(abs(dE))/(Nsample*Nclasses)
print('MSE = ' + str(MSE))
```

backpropagation

```
dense_reg.backward(dE)
ReLU.backward(dense_reg.dinputs)
dense1.backward(ReLU.dinputs)
```

optimization

```
dense_reg.weights -= alpha * dense_reg.dweights
dense_reg.biases  -= alpha * dense_reg.dbiases
dense1.weights    -= alpha * dense1.dweights
dense1.biases     -= alpha * dense1.dbiases
```

see `ANNI.ipynb` and `ANNII.ipynb`

ANN: Perceptron, Backpropagation, SGD

next time:
- fully functional ANN (minimal setup)
- regression vs classification

Thank you very much for your attention!