

Lecture 10:

ANN: Perceptron, Backpropagation, SGD



Markus Hohle

University California, Berkeley

Bayesian Data Analysis and
Machine Learning for Physical
Sciences



Course Map

Module 1	Maximum Entropy and Information, Bayes Theorem
Module 2	Naive Bayes, Bayesian Parameter Estimation, MAP
Module 3	MLE, Lin Regression
Module 4	Model selection I: Comparing Distributions
Module 5	Model Selection II: Bayesian Signal Detection
Module 6	Variational Bayes, Expectation Maximization
Module 7	Hidden Markov Models, Stochastic Processes
Module 8	Monte Carlo Methods
Module 9	Machine Learning Overview, Supervised Methods & Unsupervised Methods
Module 10	ANN: Perceptron, Backpropagation, SGD
Module 11	Convolution and Image Classification and Segmentation
Module 12	RNNs and LSTMs
Module 13	RNNs and LSTMs + CNNs
Module 14	Transformer and LLMs
Module 15	Graphs & GNNs

Outline



- Gradient Descent

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

- Perceptron

- Backpropagation

Outline

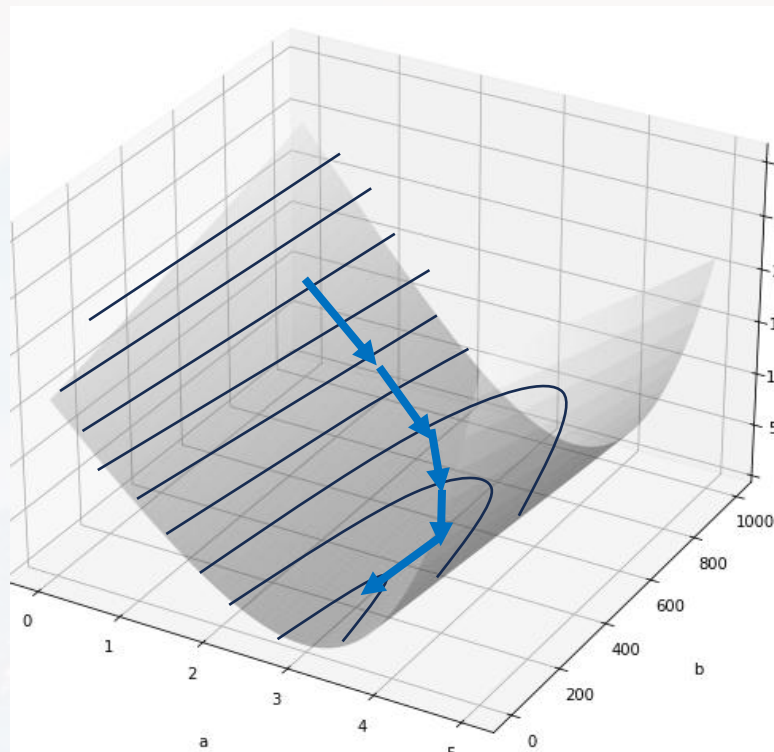
- Gradient Descent

- 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 \rightarrow leads to oscillations
 ϵ too small \rightarrow 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**)



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

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

AdaptiveGradient:

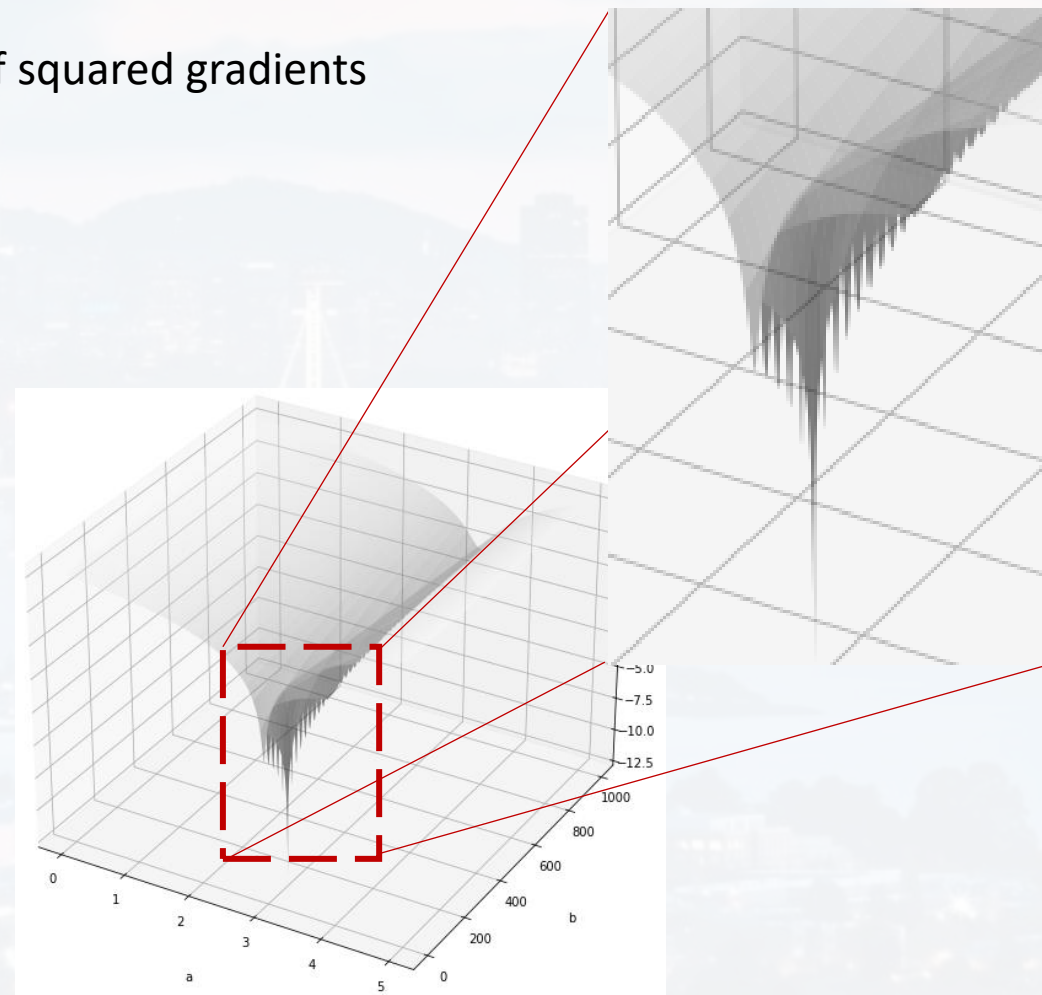
idea: reducing ϵ over time by **accumulated sum** of squared gradients

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

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

$\delta > 0$ 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

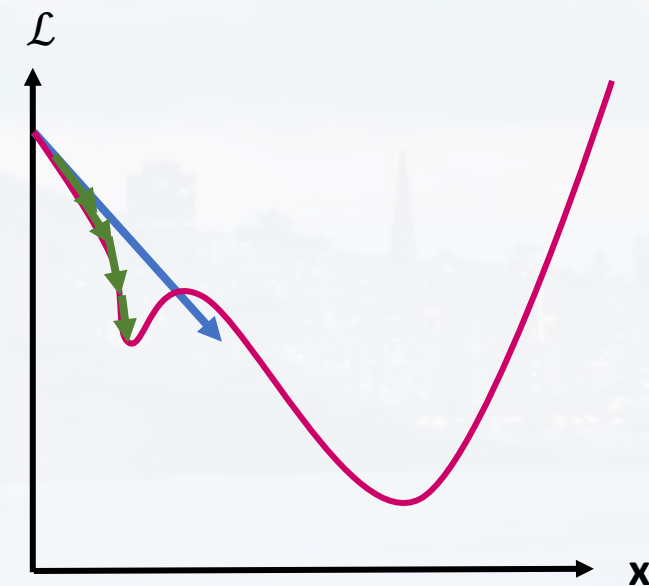
$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_t) \rangle = \frac{1}{N} [\nabla_{\vec{x}} \mathcal{L}(\vec{x}_{t-1}) + \nabla_{\vec{x}} \mathcal{L}(\vec{x}_{t-2}) + \dots + \nabla_{\vec{x}} \mathcal{L}(\vec{x}_{t-N})]$$

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





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

$\epsilon > 0$:	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

$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0) \rangle = \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0)$$

$$\mu_0 = (0,1)$$

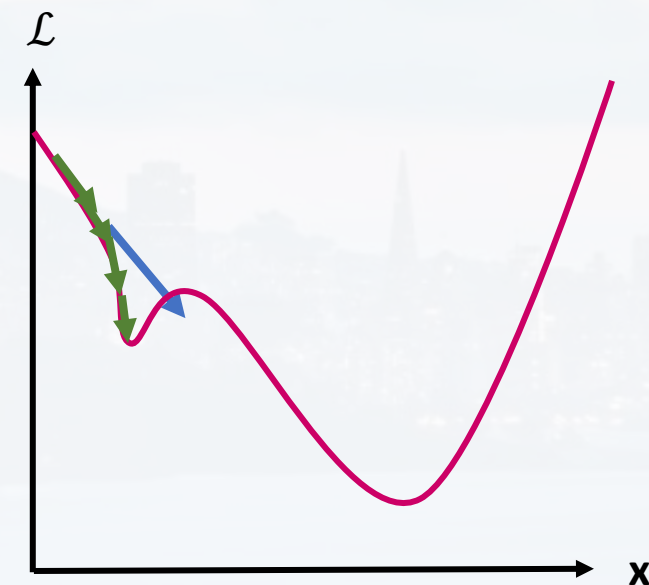
$$\langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) \rangle = \nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) + \mu_0 \cdot \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0)$$

$$\begin{aligned} \langle \nabla_{\vec{x}} \mathcal{L}(\vec{x}_2) \rangle &= \nabla_{\vec{x}} \mathcal{L}(\vec{x}_2) + \boxed{\mu_0} [\nabla_{\vec{x}} \mathcal{L}(\vec{x}_1) \\ &\quad + \boxed{\mu_0} \nabla_{\vec{x}} \mathcal{L}(\vec{x}_0)] \end{aligned}$$

$$\mu_{k=2} = \mu_0 \mu_0 = \mu_0^2$$

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

... and so on...





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

momentum:

weighted average with weighting factor μ_k

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

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

... and so on...

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
```

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



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

regularization: recall: linear regression

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

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

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

L1 or **Least absolute shrinkage and selection operator**
- encourages **sparsity** of β
- reduces **overfitting**

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

L2 or **Ridge**
- **penalizes large β**

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

...and so on



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

regularization:

L1 and L2 regularization

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

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

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



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

regularization:

L1 and L2 regularization

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

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

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

- 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



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

$\epsilon > 0$:	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 \rightarrow slows down updates too fast

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

idea: weighted average like for **momentum**

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

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

$$\epsilon \rightarrow \frac{\epsilon}{\sqrt{r_{t+1}} + \delta}$$

adaptive gradient, aka **AdaGrad**

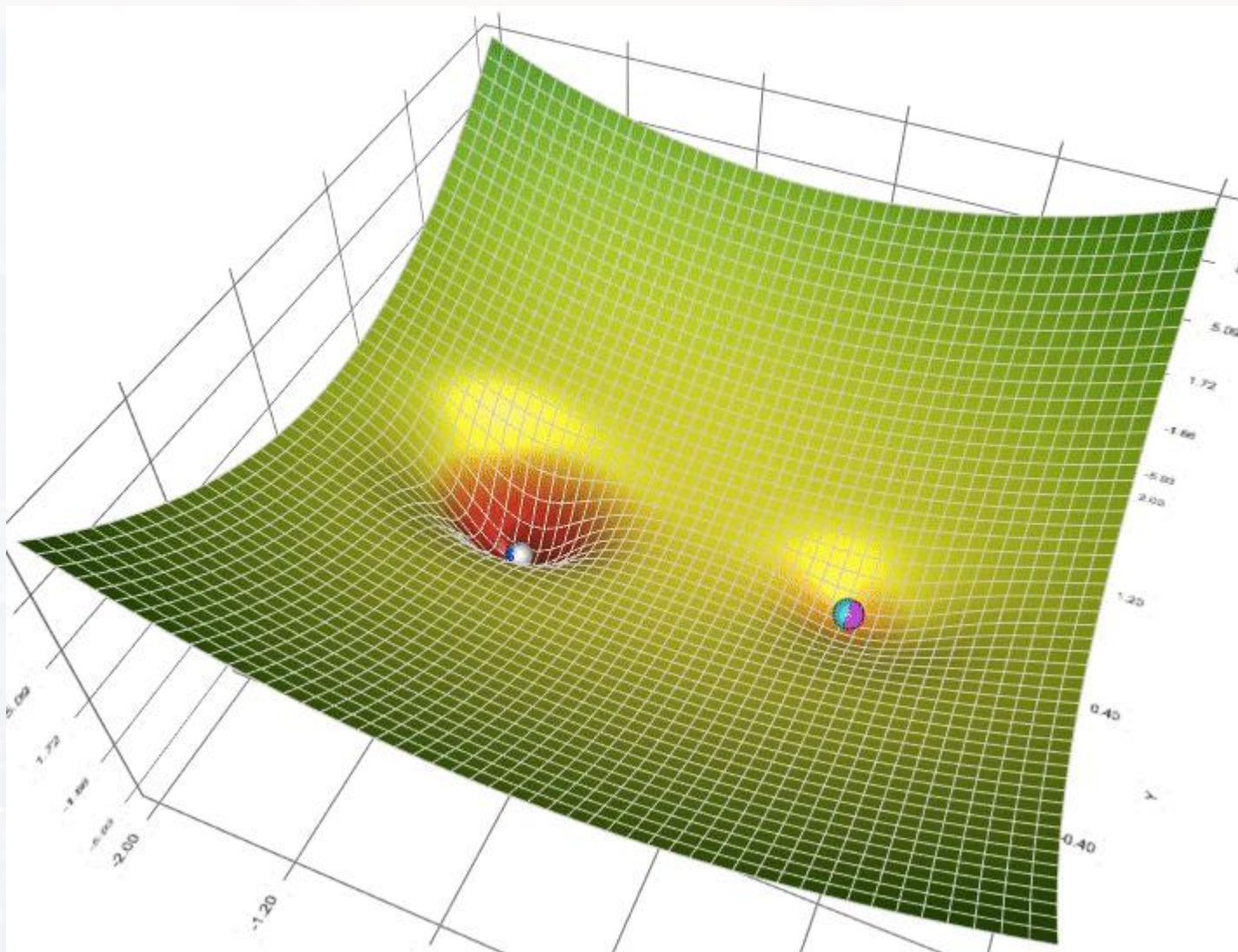
Root Mean Square Propagation **RMSProp**

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



Lili Jiang

[TowardsDataScience](#)

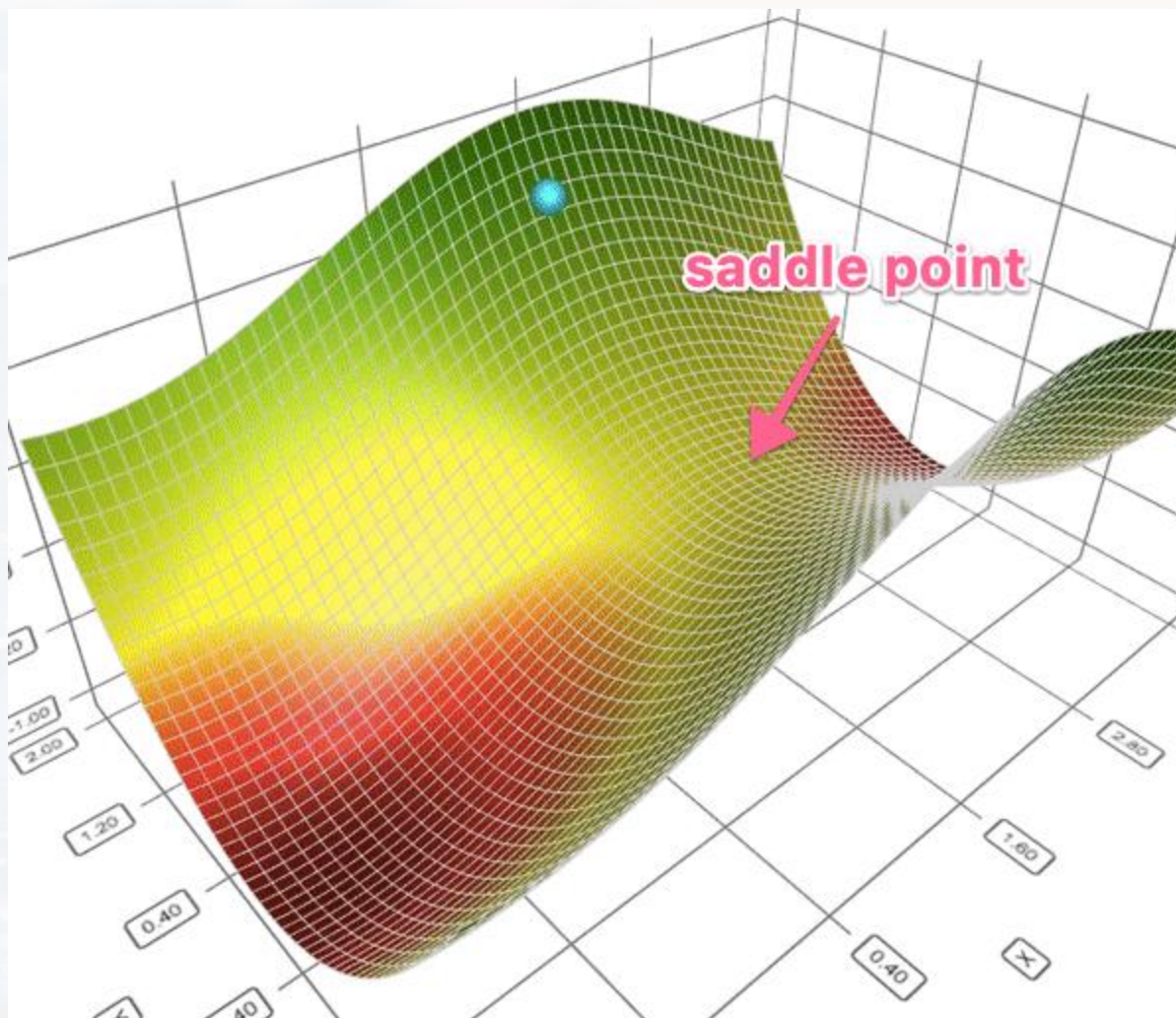


gradient descent (cyan),
momentum (magenta),
AdaGrad (white),
RMSProp (green),
Adam (blue)



Lili Jiang

[TowardsDataScience](#)



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

see also
`WalkThroughGradDescent.ipynb`

Outline

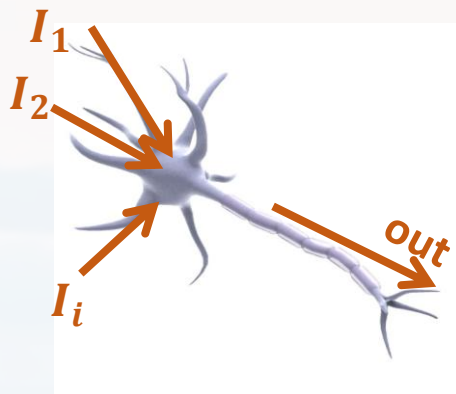


- Gradient Descent

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

- Perceptron

- Backpropagation



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

weights and bias are
“**learnable**”

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

recall: linear models

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

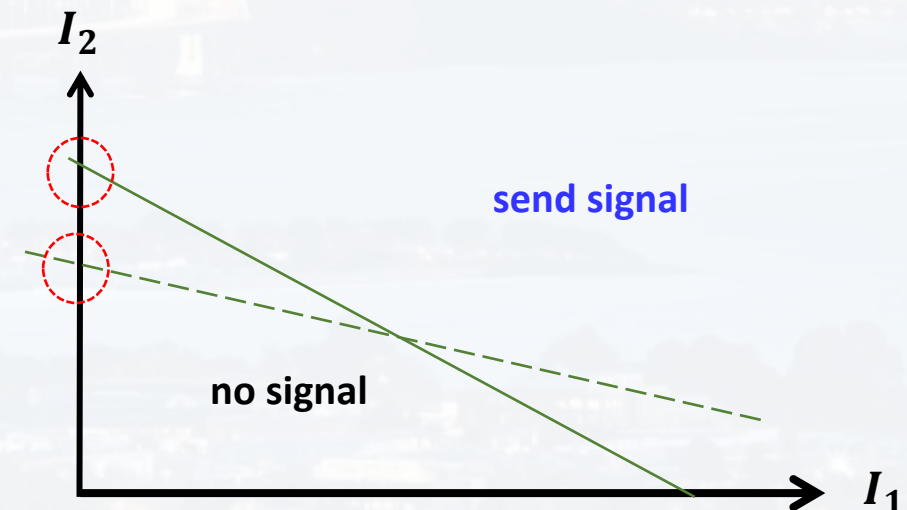
$$\begin{pmatrix} y_1 \\ \vdots \\ y_k \\ \vdots \\ y_K \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & x_{12} & \dots & x_{1n} & \dots & x_{1N} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \\ 1 & x_{k1} & & & x_{kn} & & \\ \vdots & \vdots & & & \vdots & & \vdots \\ 1 & \dots & & & \dots & & \\ 1 & x_{K1} & x_{K2} & \dots & x_{Kn} & \dots & x_{KN} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \\ \vdots \\ \beta_N \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_k \\ \vdots \\ \epsilon_K \end{pmatrix}$$

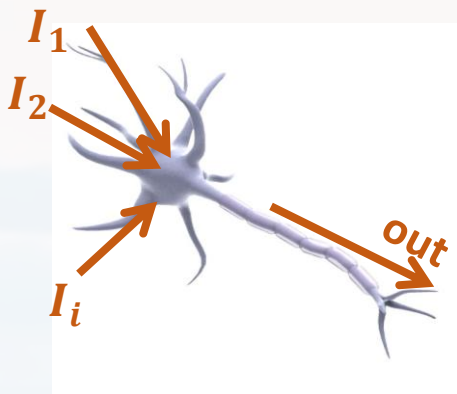
simple example:

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

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

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

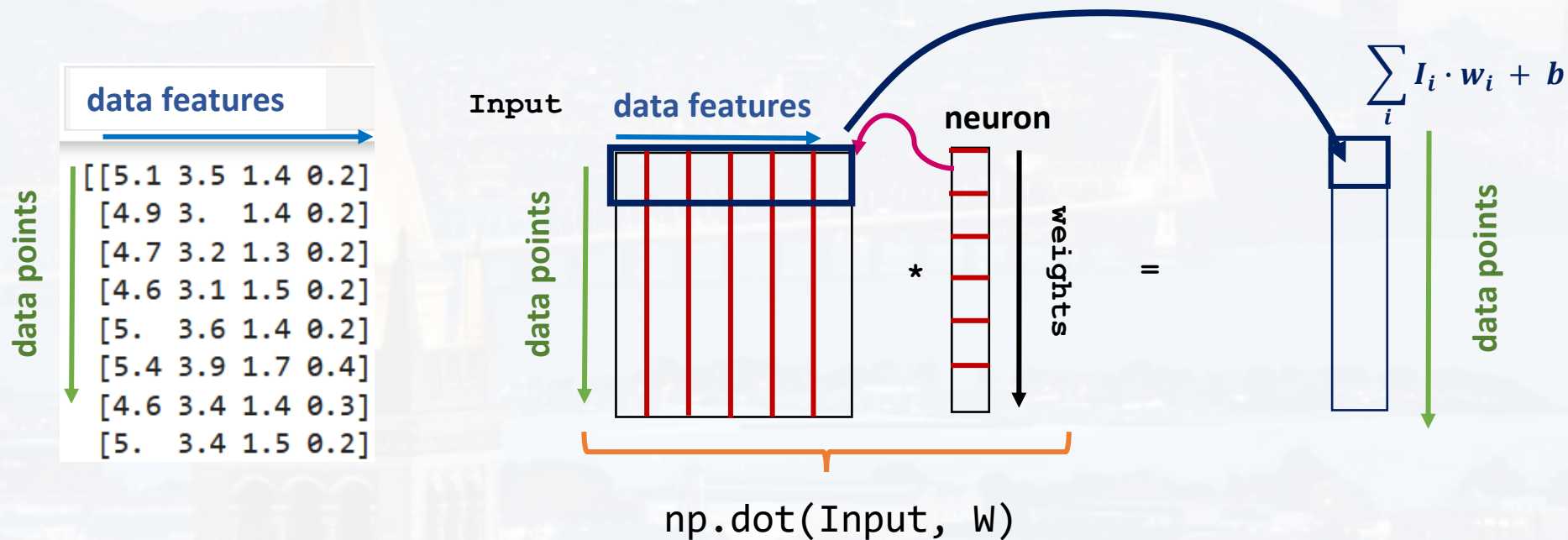


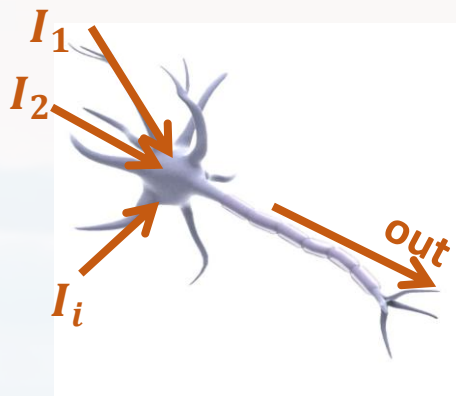


$$net = \sum_i I_i \cdot w_i + b$$

weights and bias are
“learnable”

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**”

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

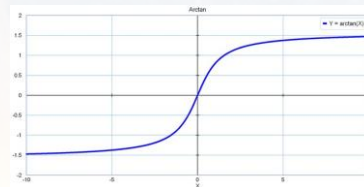
net →

some activation function f

→

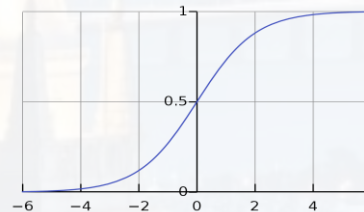
actual output

- $y = \arctan(net)$



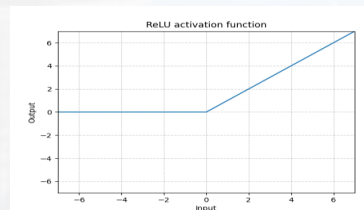
$(-\infty; +\infty) \rightarrow (-\pi/2; +\pi/2)$

- $y = \text{sigm}(net)$



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

- $y = \text{ReLU}(net)$
 $= \max(0, net)$



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

...and many other

Outline



- Gradient Descent

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

- Perceptron

- **Backpropagation**



learning:

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

target output t

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

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

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

outer derivatives inner derivative

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

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

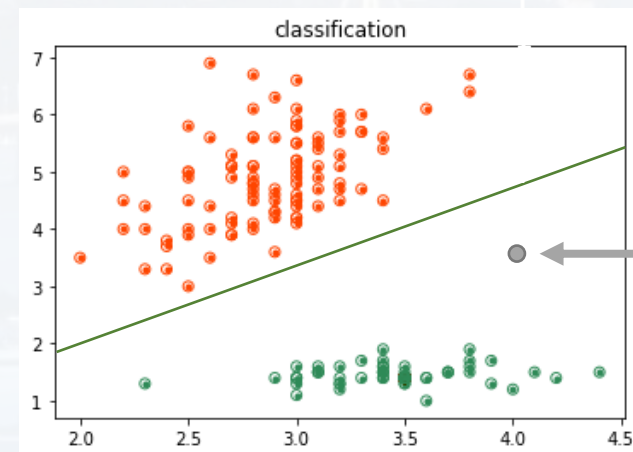
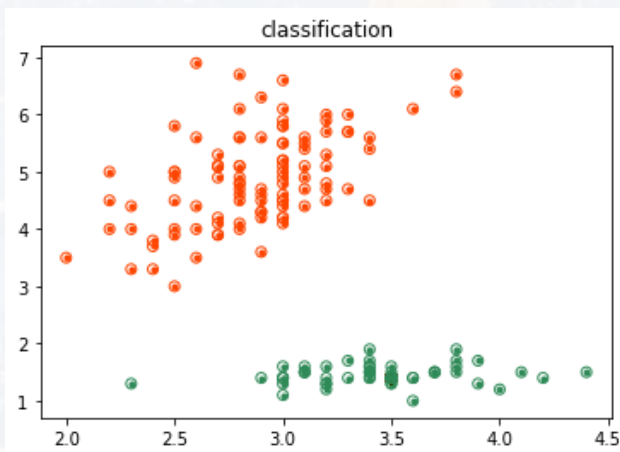
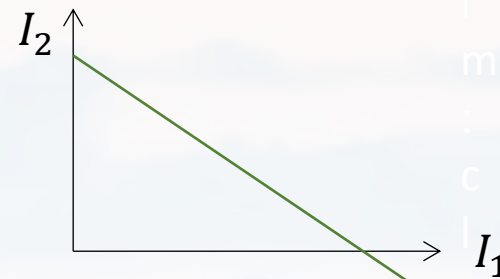
$$\Delta b = b(new) - b(old) = \epsilon (t - y) f'(net) \cdot 1$$

see Perceptron.ipynb



see `Perceptron.ipynb`

one neuron
two features
two classes
N data points

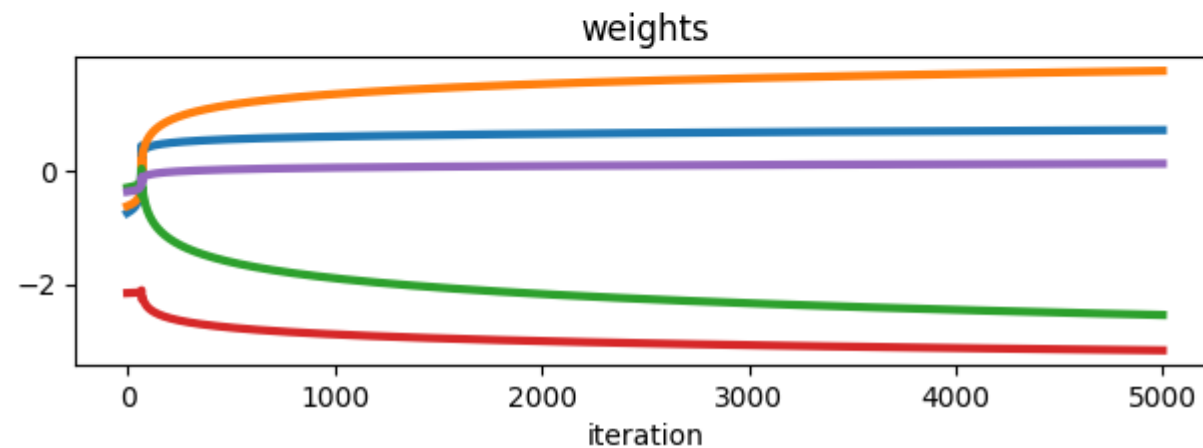
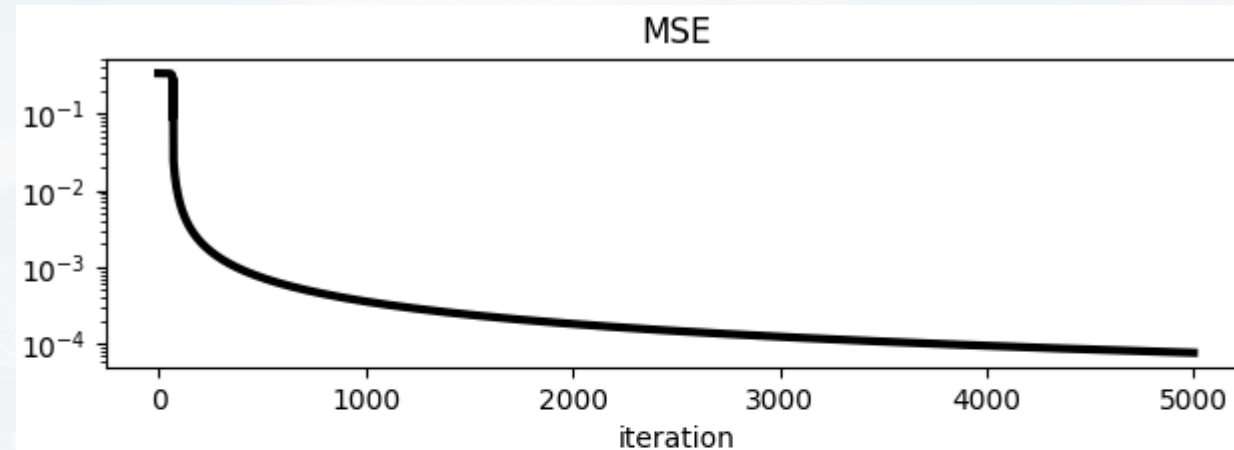
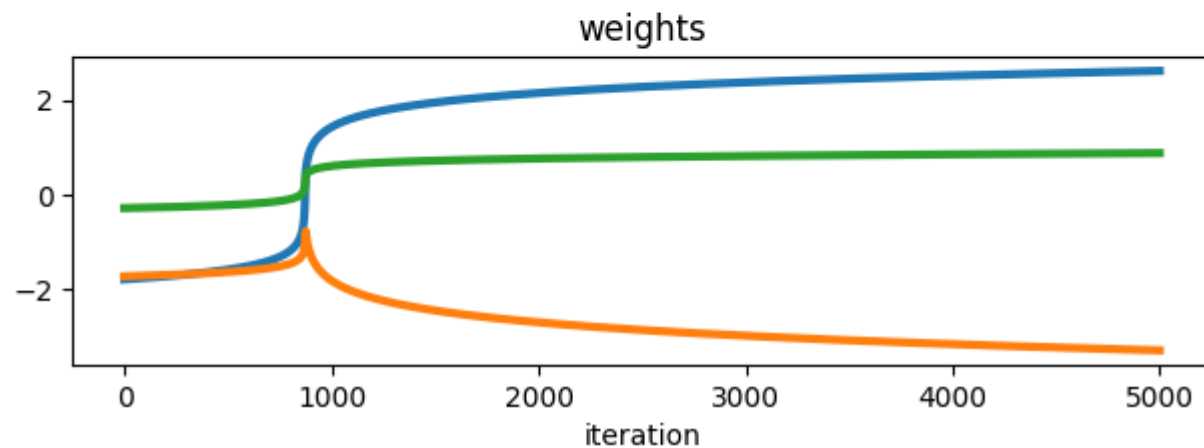
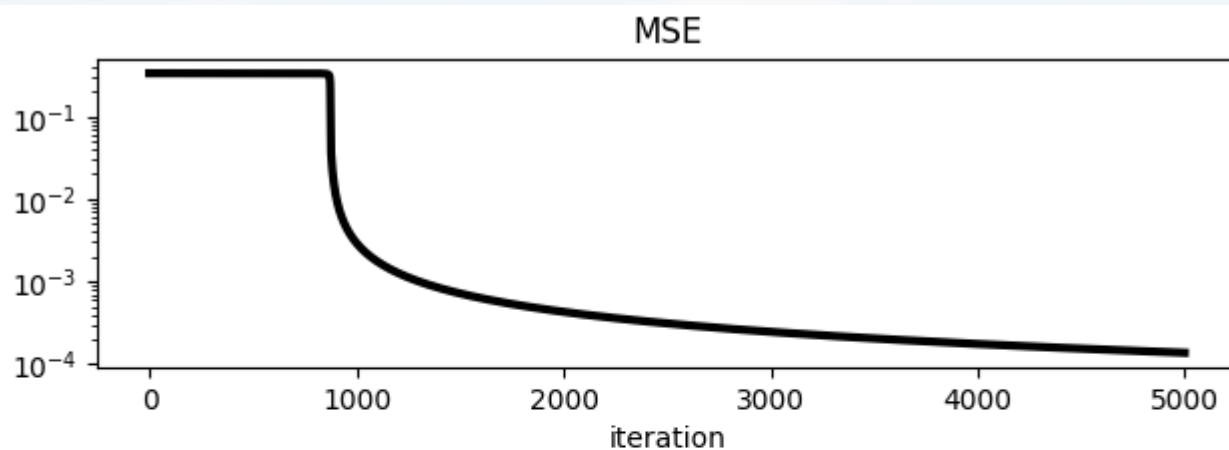


classifying a new
data point



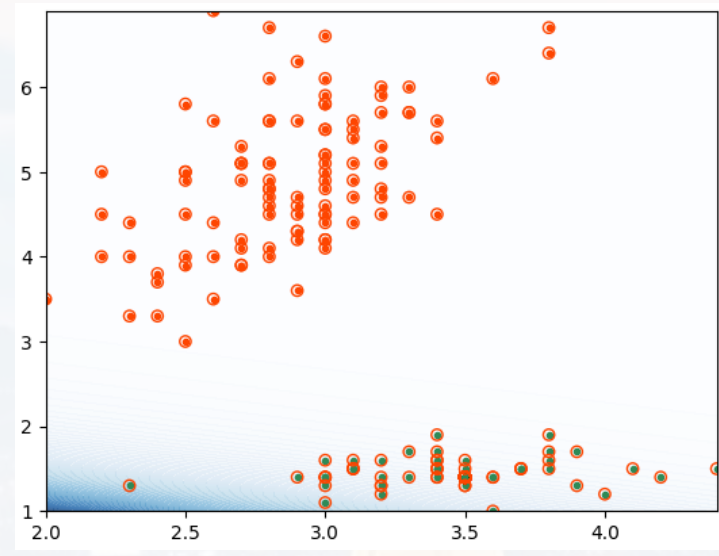
see `Perceptron.ipynb`

the training process

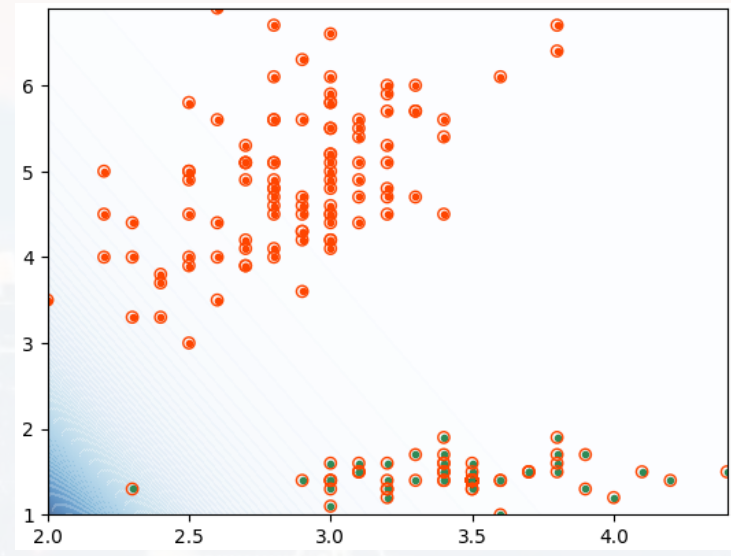


see `Perceptron.ipynb`

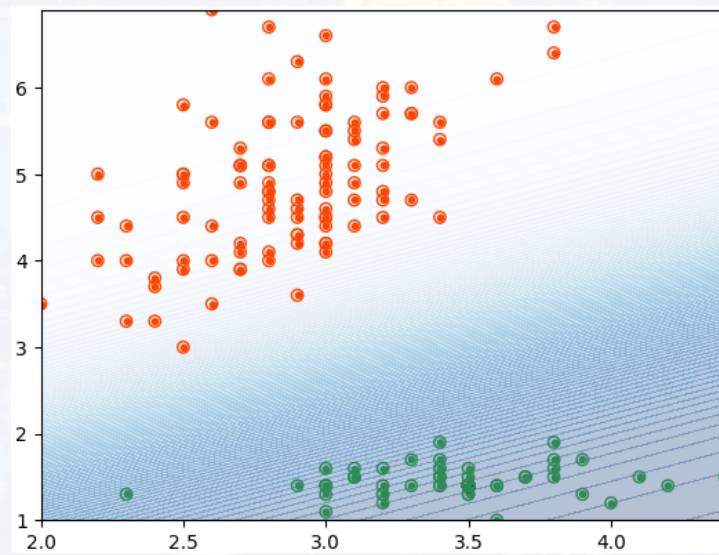
the training process



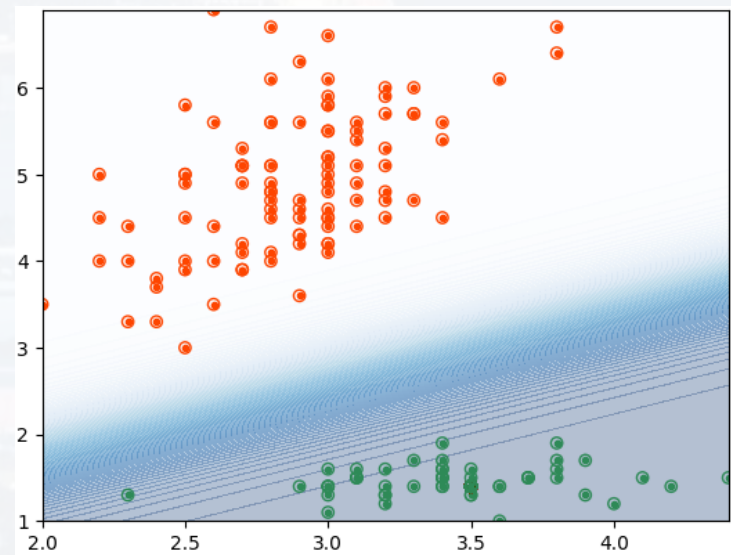
t = 3



t = 5



t = 100



t = 10,000



dense layer:

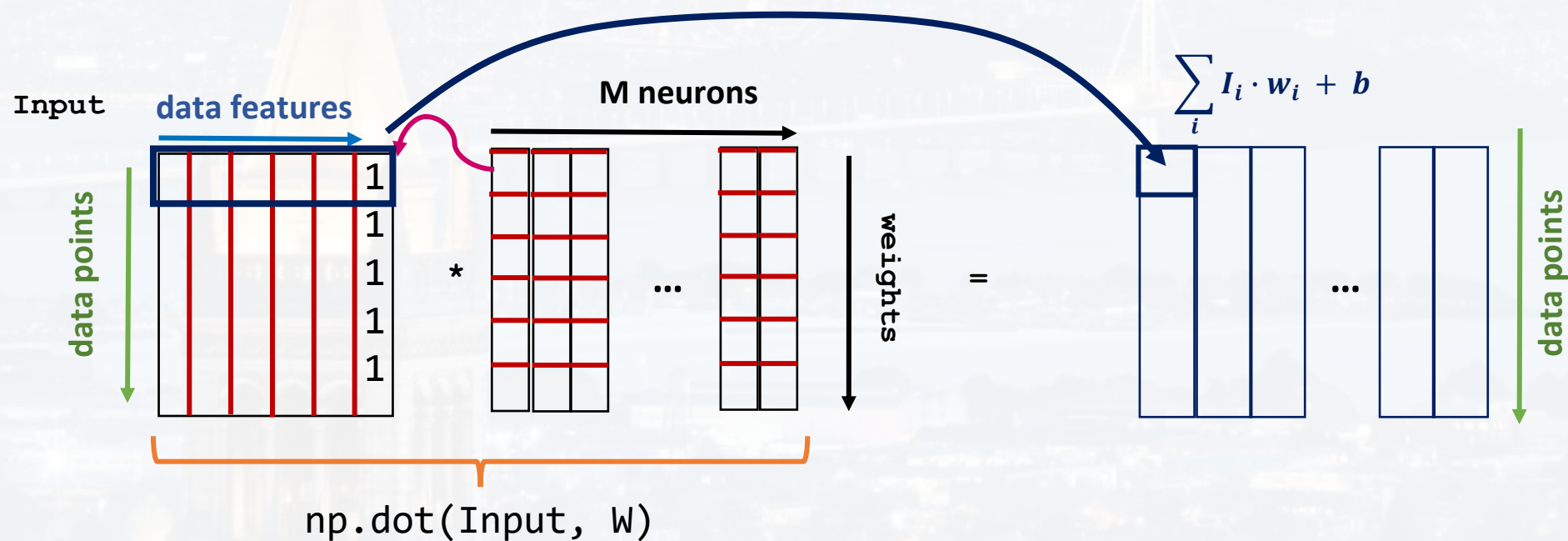
multiple neurons

data features

data points

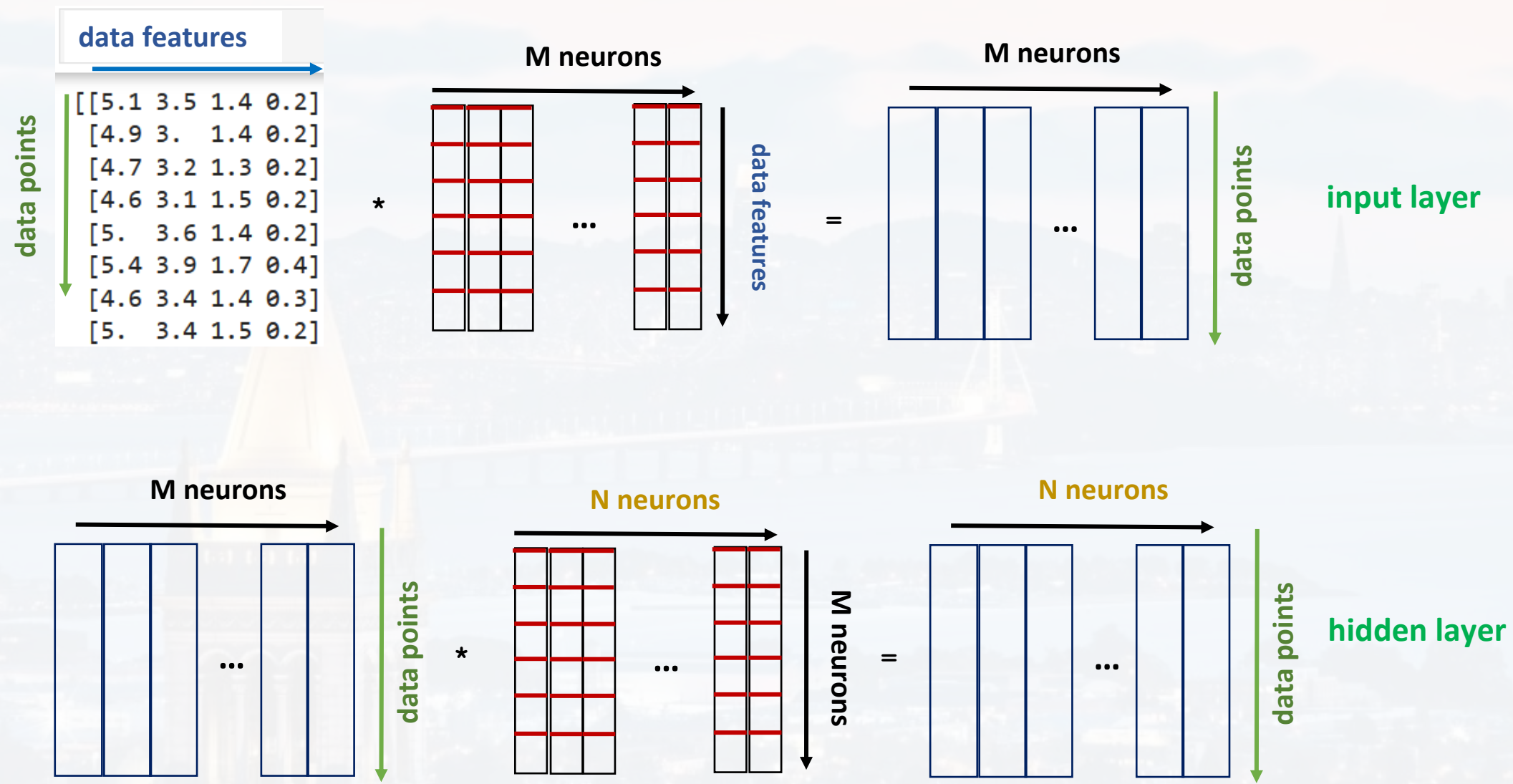
[5.1	3.5	1.4	0.2]
[4.9	3.	1.4	0.2]
[4.7	3.2	1.3	0.2]
[4.6	3.1	1.5	0.2]
[5.	3.6	1.4	0.2]
[5.4	3.9	1.7	0.4]
[4.6	3.4	1.4	0.3]
[5.	3.4	1.5	0.2]

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

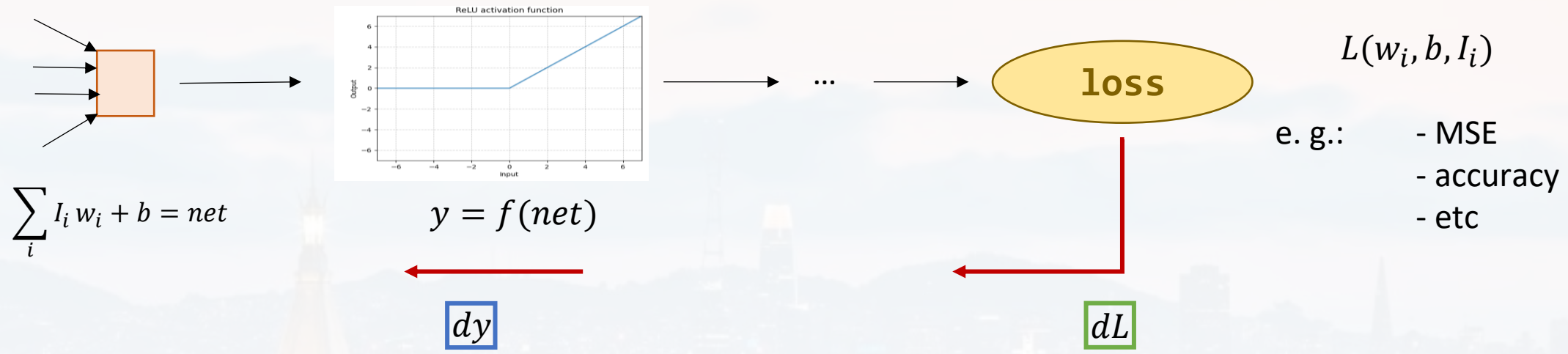


adding subsequent dense layer

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



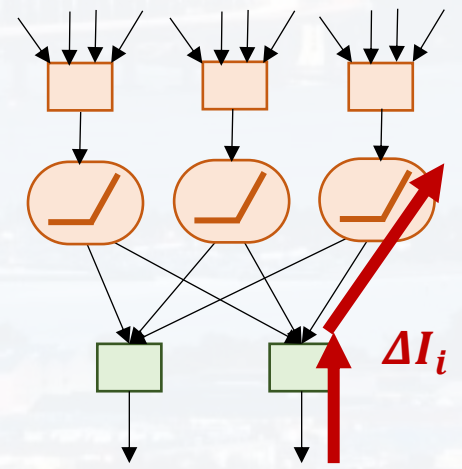
for training: building the backpropagation part



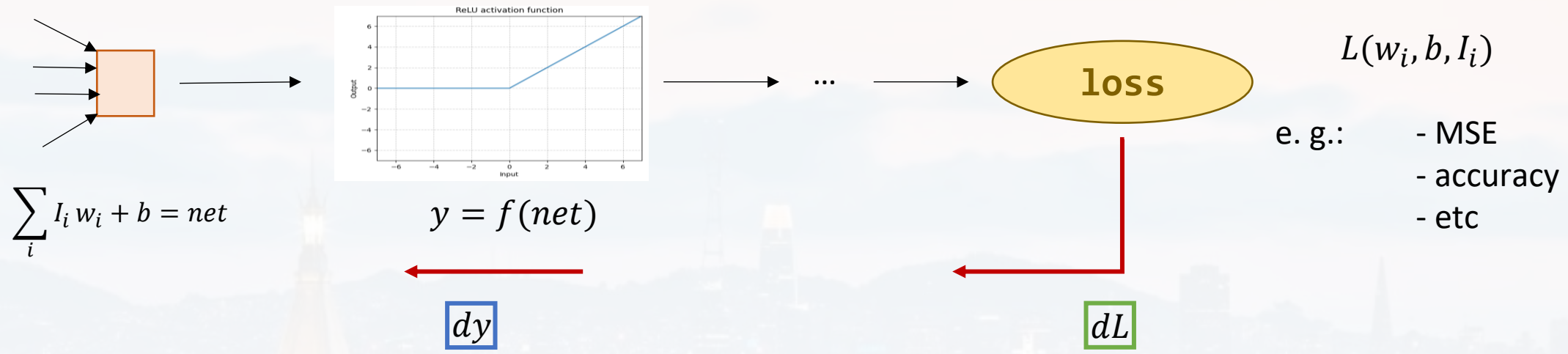
$$\Delta w_i = -\epsilon \frac{dL}{dy} \frac{dy}{dnet} I_i$$

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

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



for training: building the backpropagation part



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

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

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

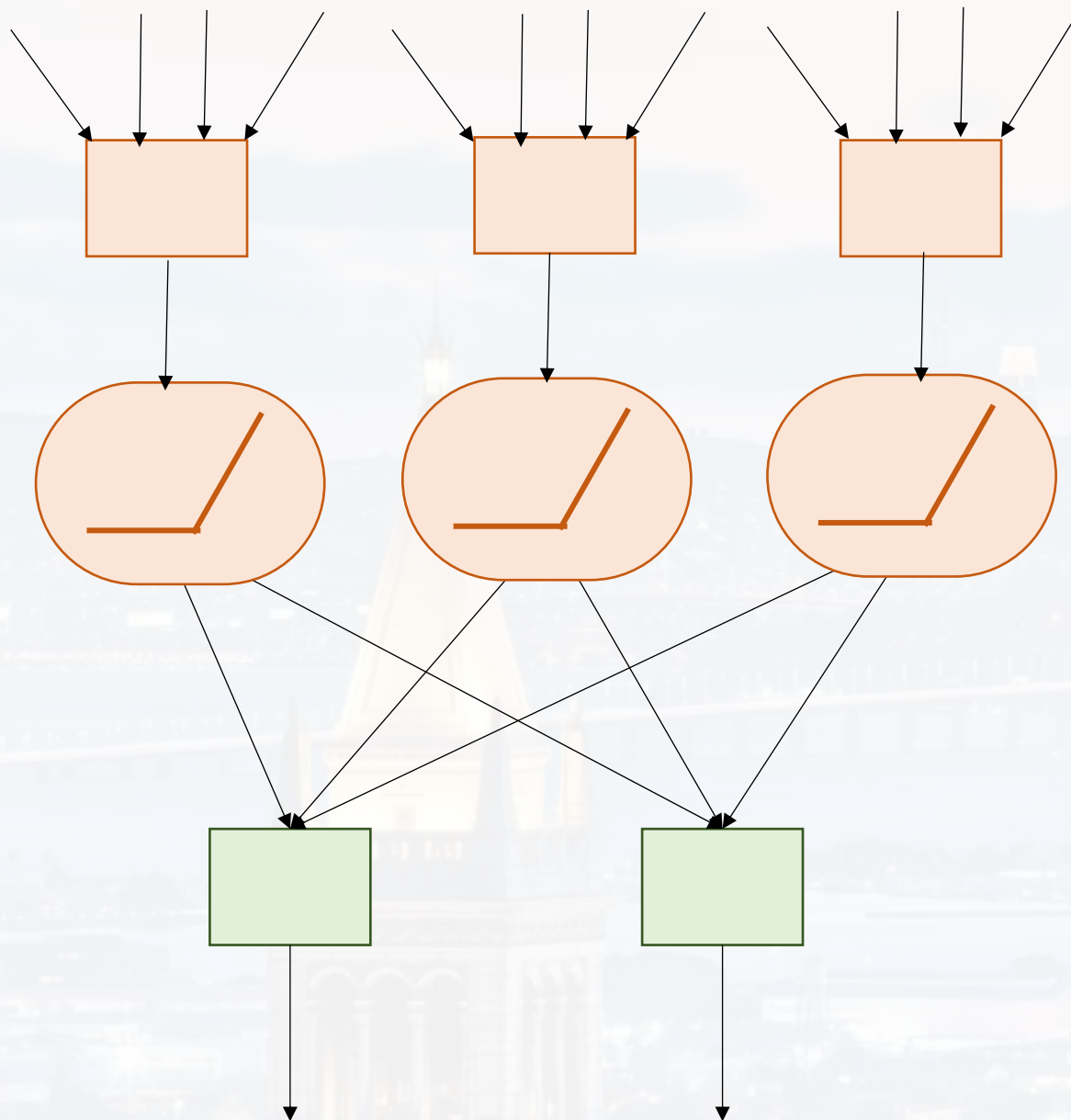
hence, we need to include the following structure for backpropagation:

```

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

inner
derivative

product of
outer
derivatives

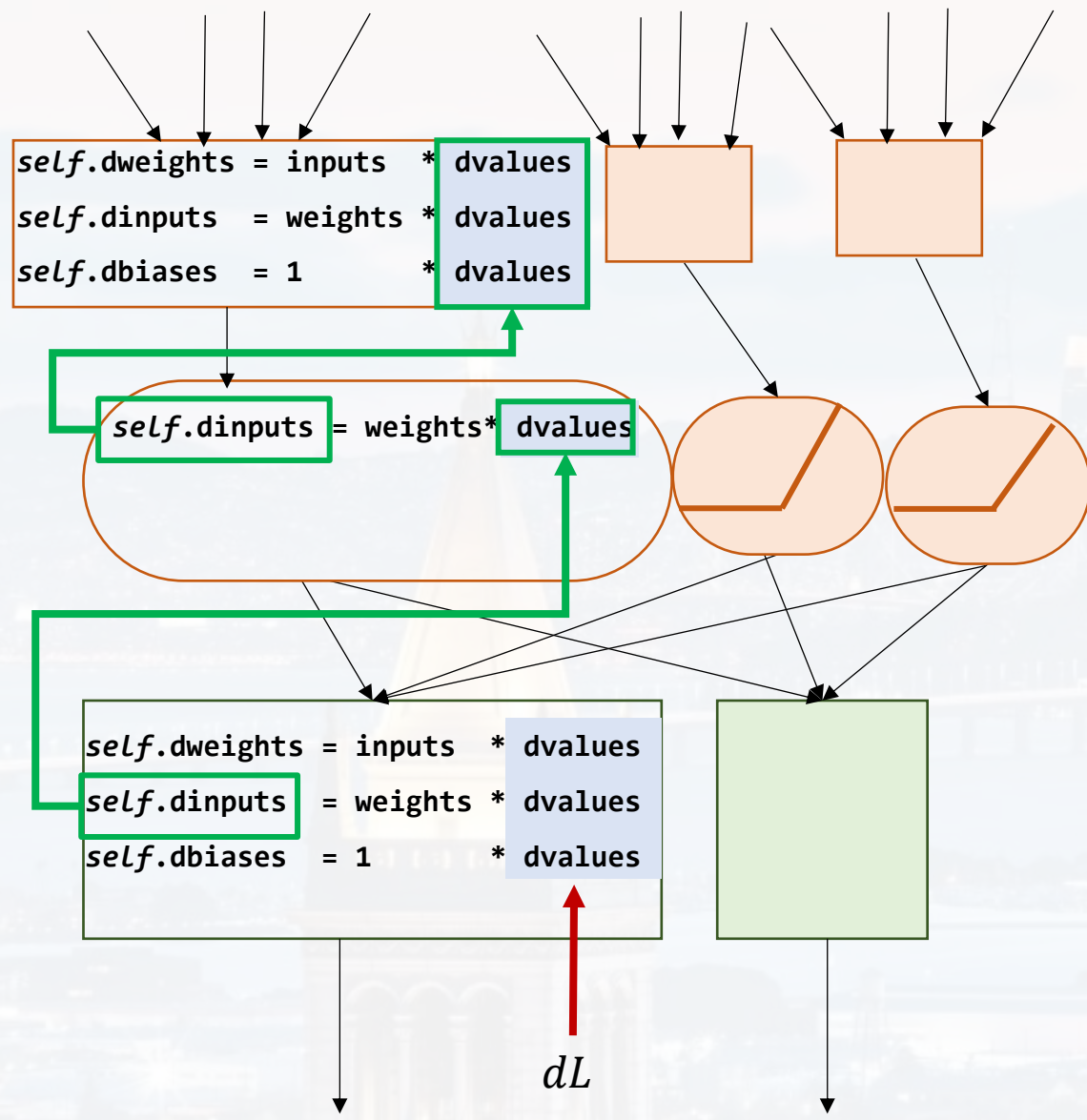


hence, we need to include the following structure for backpropagation:

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

inner
derivative

product of
outer
derivatives



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

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

$$\Delta I_i = -\epsilon \left[\frac{dL}{dy} \right] \left[\frac{dy}{dnet} \right] w_i$$

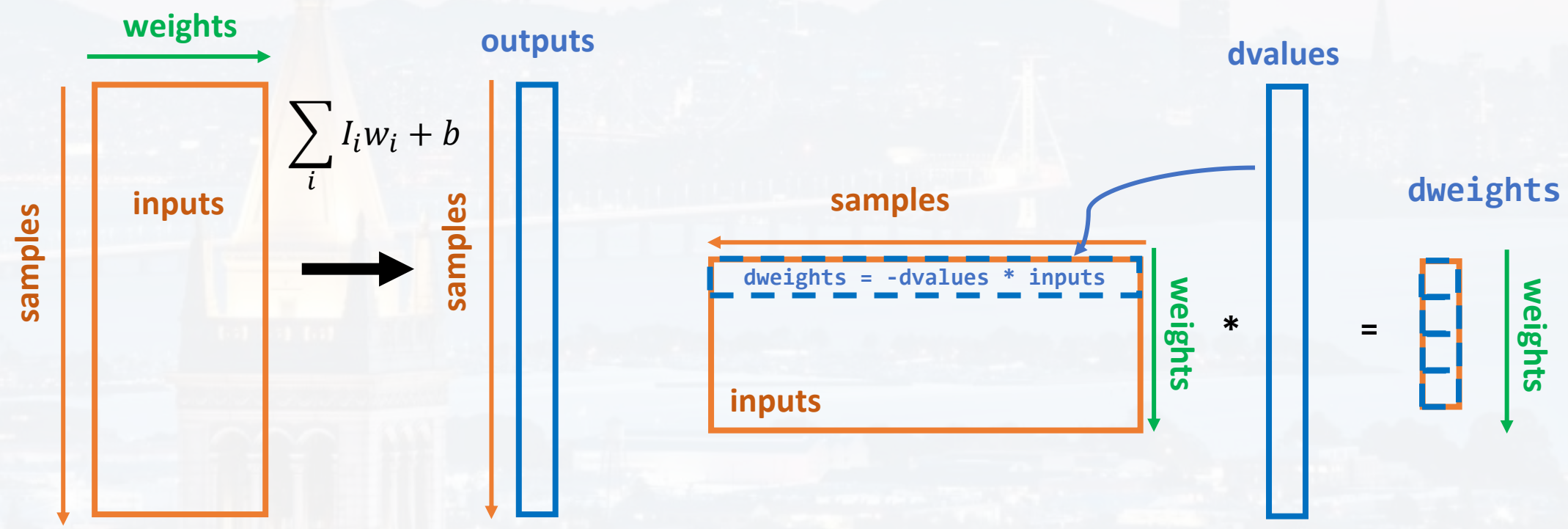
We need a **forward** part and a **backward** part!

single neuron:

forward

backward

$$\text{dweights} = \text{dvalues} * \text{inputs}$$





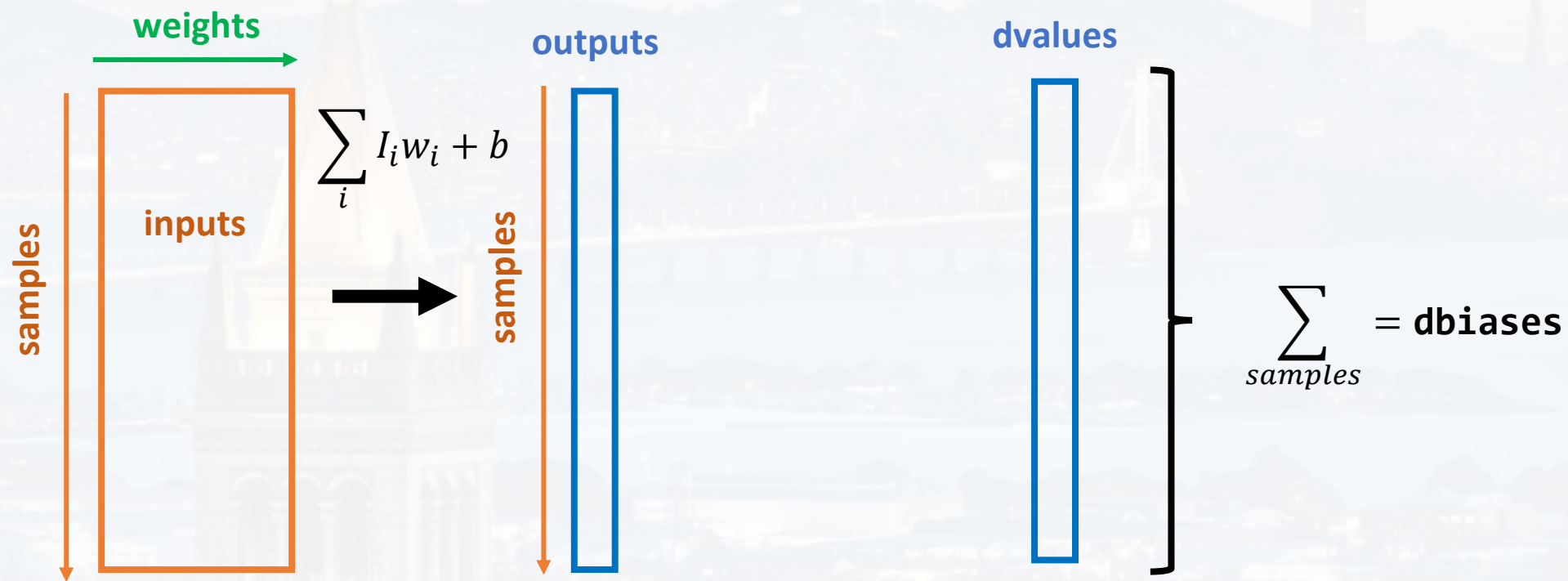
We need a **forward** part and a **backward** part!

single neuron:

forward

backward

dbiases = **d**values

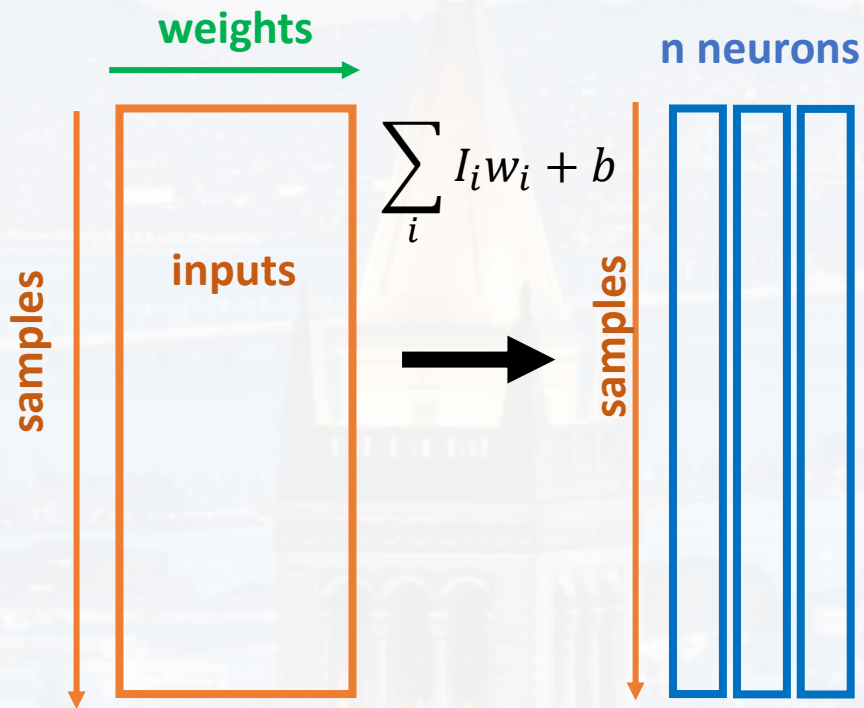




We need a **forward** part and a **backward** part!

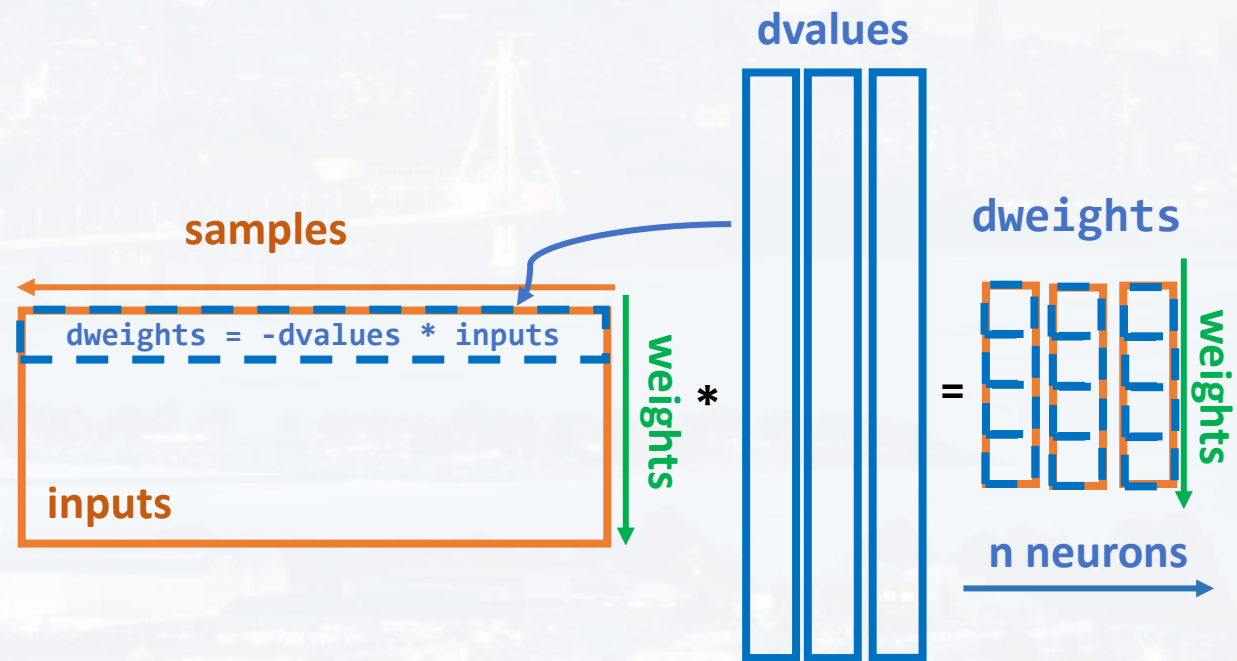
dense layer (say **three** neurons):

forward



backward

$$\text{dweights} = \text{dvalues} * \text{inputs}$$

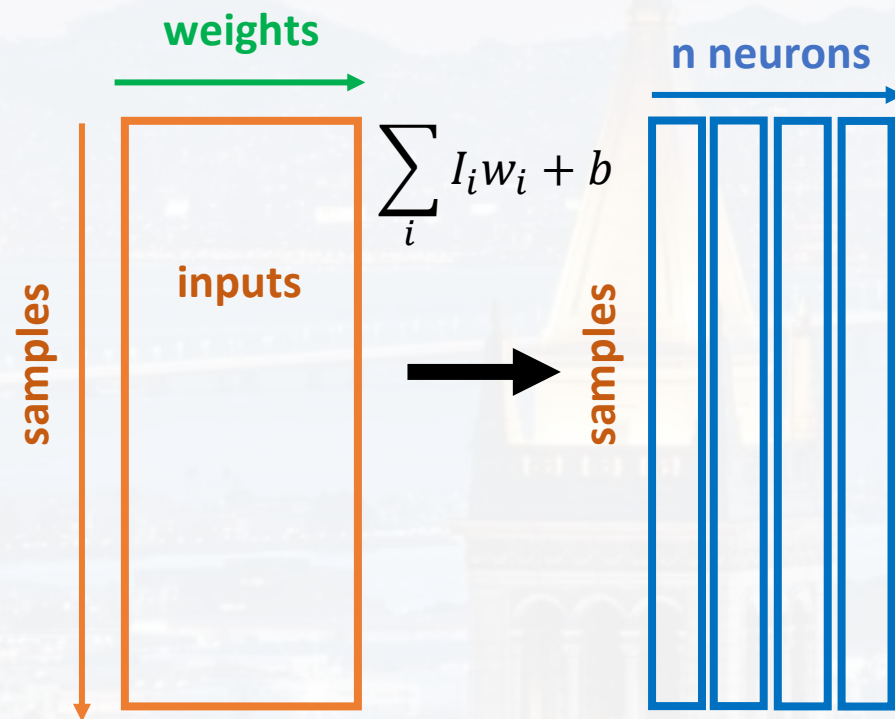




We need a **forward** part and a **backward** part!

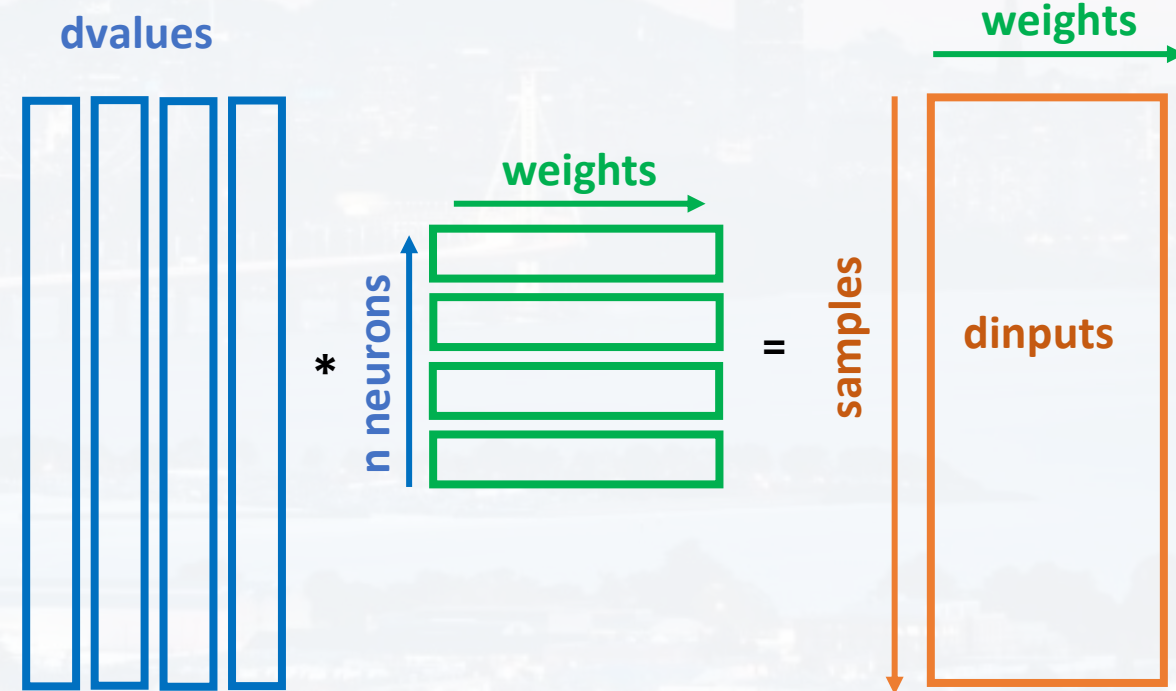
dense layer (say **four** neurons):

forward



backward

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





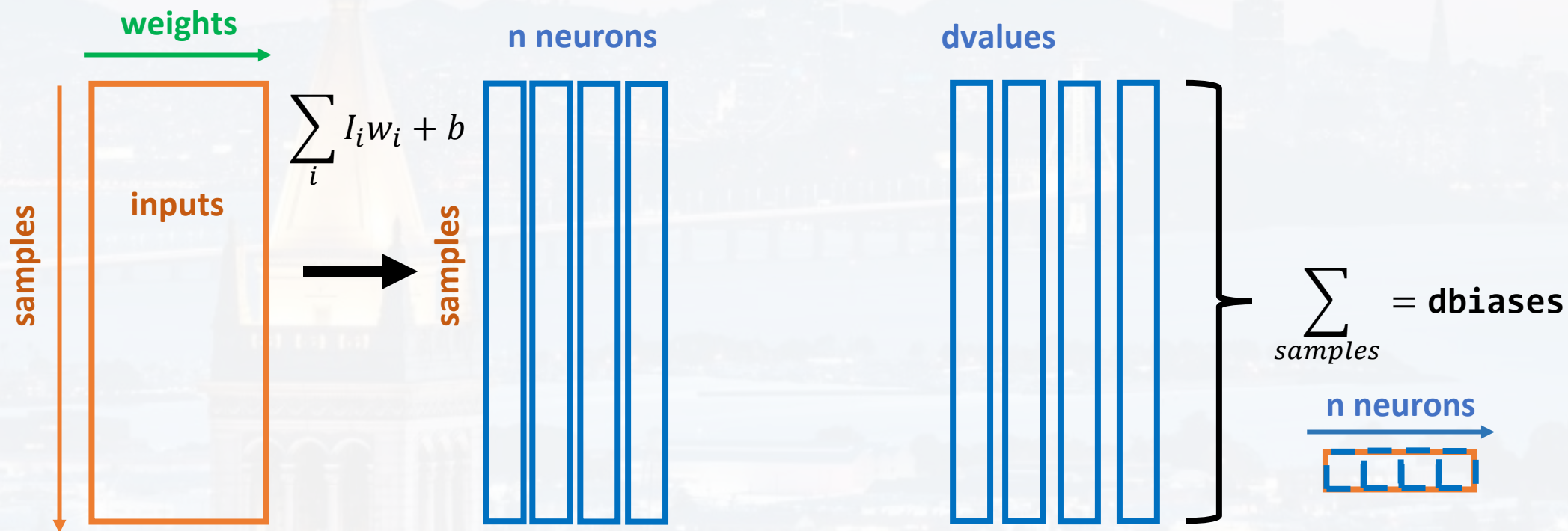
We need a **forward** part and a **backward** part!

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

forward

backward

dbiases = dvalues





```
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,))
```

number of features

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

outer derivative

```
    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)
```

<code>self.dweights</code>	<code>= inputs</code>	<code>*</code>	<code>dvalues</code>
<code>self.dinputs</code>	<code>= weights</code>	<code>*</code>	<code>dvalues</code>
<code>self.dbiases</code>	<code>= 1</code>	<code>*</code>	<code>dvalues</code>

inner derivative

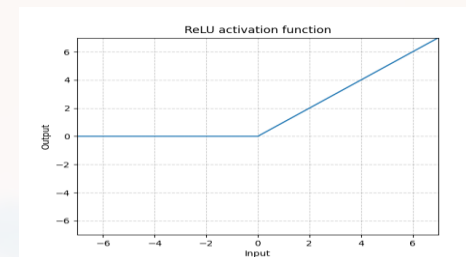
product of outer derivatives



```
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
```



```
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`

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

forward

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

evaluation

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

backpropagation

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

optimization

see ANNI.ipynb and ANNII.ipynb

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

Thank you very much for your attention!

