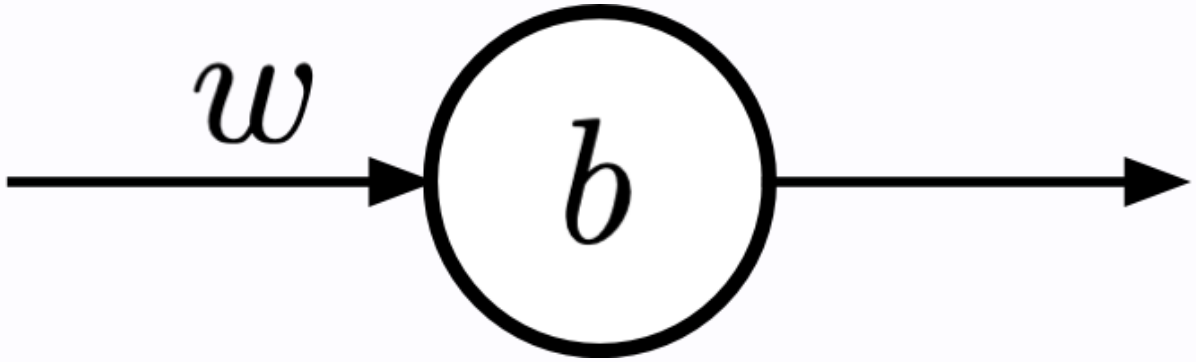
Improving the Way Neural Networks Learn

Up to now we've focused on understanding the backpropagation algorithm. In the following we will introduce a suite of techniques which can be used to improve the way the networks learn.

The Problem of Slow Learning

Networks based on sigmoid units with a quadratic loss function, often learn very slowly.

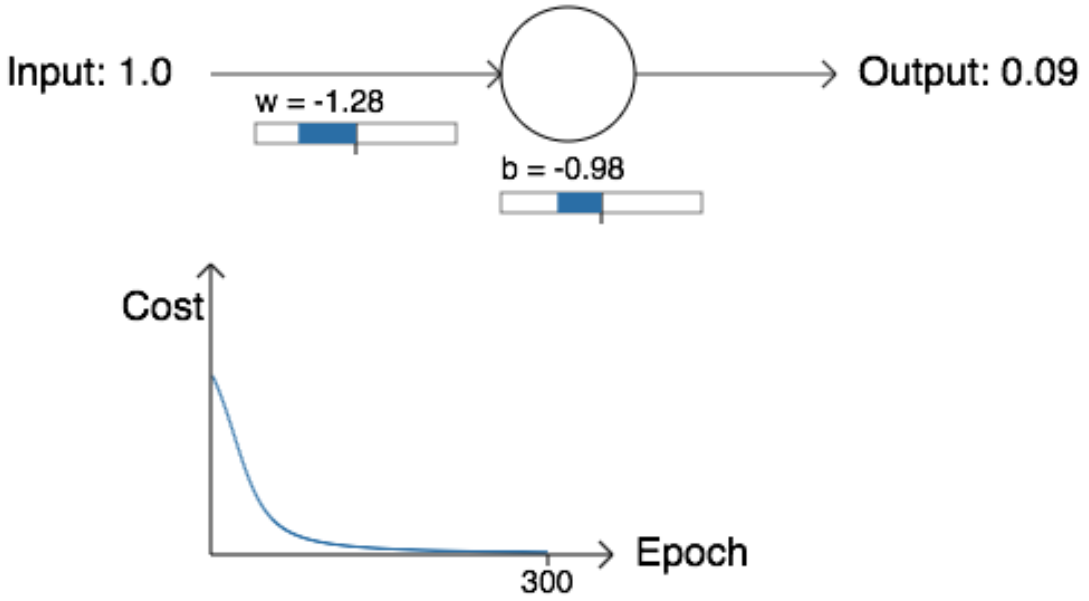


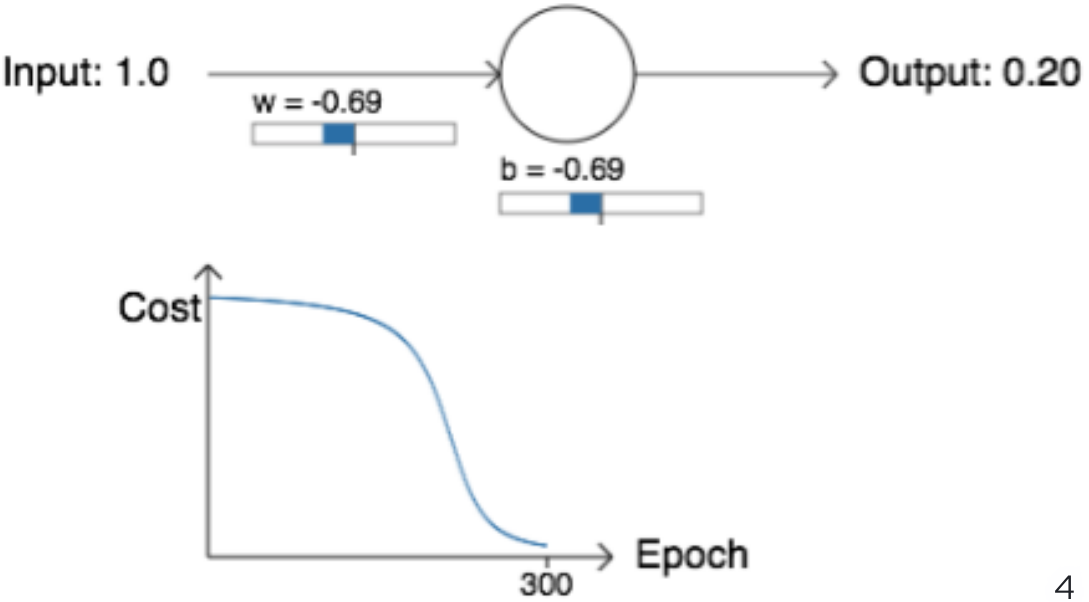
To understand why let us consider the very simple neural network on the right and consider the problem of learning to map input 1 to output 0. In this example C is the quadratic cost and the output unit is a sigmoid.

The Problem of Slow Learning

First graph: the graph of the cost function when the weights are initialized to $w = 0.6$ and $b = 0.9$.

Second graph: the graph of the cost function when the weights are initialized to $w = 2.0$ and $b = 2.0$.





The Problem of Slow Learning

Why does this happen? Let us recall that for this particular case $C = \frac{(y-z)^2}{2}$, and $z = \sigma(a) = \sigma(wx + b)$. With this in mind, it is easy to see that:

$$\frac{\partial C}{\partial w} = (z - y)\sigma'(a)x = z\sigma'(a)$$

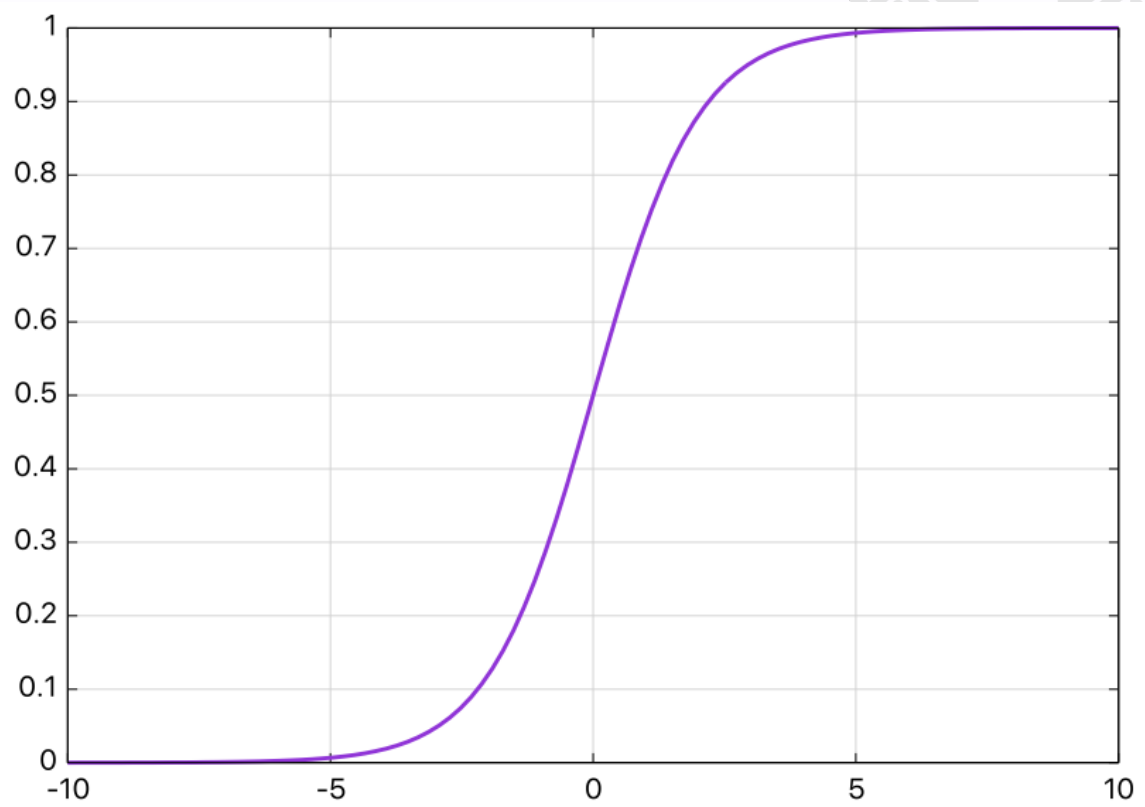
and

$$\frac{\partial C}{\partial b} = (z - y)\sigma'(a) = z\sigma'(z)$$

where in both formulas the second equality is obtained by substituting $x = 1$ and $y = 0$.

The Problem of Slow Learning

The problem in our second example is that by choosing $w = 2.0$ and $b = 2.0$ as initial values of the network parameters, we start our learning in a place where $z = 2 \times 1 + 2 = 4$ and at that point the sigmoid function is very flat and so its derivative is almost zero.



Cross-Entropy

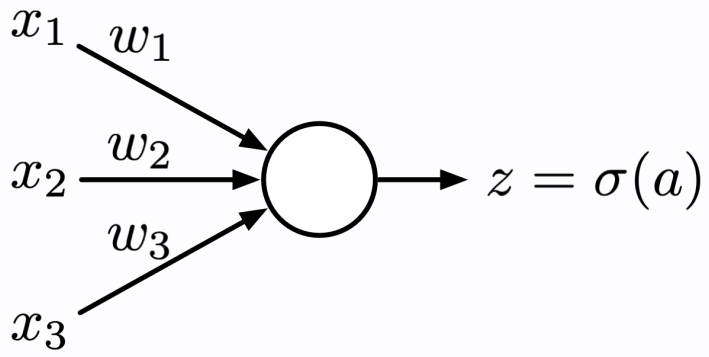
The cross-entropy cost function solves the slow-learning problem without changing the activation function (i.e., σ will still be defined to be the sigmoid function).

The cross-entropy is defined as:

$$C = -\frac{1}{n} \sum_{\mathbf{x}} [y \ln z + (1-y) \ln(1-z)]$$

where, as usual, y is the desired output and z is the activation of the neuron.

Note: as shown in the image, we are still considering a single neuron network, but in the slightly more general case where it has more than one input: $a = \sum_j w_j x_j + b$.



Cross-Entropy

Note that C still behave as a cost function:

- if $y = 1$ and $z \approx 1$, then:

$$-(y \ln z + (1 - y) \ln(1 - z)) \approx 0$$

- if $y = 1$ and $z \approx 0$, then:

$$-(y \ln z + (1 - y) \ln(1 - z)) \approx \infty$$

The same goes for $y = 0$ and $z \approx 0$ and $y = 0$ and $z \approx 1$.

Summing up: the cross entropy function is always positive and tend to zero as the computed outputs tend to the desired outputs.

Cross Entropy: $C = -\frac{1}{n}\sum_{\mathbf{x}}[y \ln z + (1-y)\ln(1-z)]$

$$
\begin{aligned}
\frac{\partial C}{\partial w_j} &= -\frac{1}{n}\sum_{\mathbf{x}} \frac{y}{\sigma(a)}\frac{\partial \sigma}{\partial w_j} - \frac{1-y}{1-\sigma(a)}\frac{\partial \sigma}{\partial w_j} \\
&= -\frac{1}{n}\sum_{\mathbf{x}} \left(\frac{y}{\sigma(a)} - \frac{1-y}{1-\sigma(a)}\right)\sigma'(a)x_j \\
&= -\frac{1}{n}\sum_{\mathbf{x}} \left(\frac{(1-\sigma(a))y - (1-y)\sigma(a)}{\sigma(a)(1-\sigma(a))}\right)\sigma'(a)x_j \\
&= -\frac{1}{n}\sum_{\mathbf{x}} \left(\frac{y - y\sigma(a) - \sigma(a) + y\sigma(a)}{\sigma(a)(1-\sigma(a))}\right)\sigma'(a)x_j \\
&= -\frac{1}{n}\sum_{\mathbf{x}} \left(\frac{y - \sigma(a)}{\sigma(a)(1-\sigma(a))}\right)\sigma'(a)x_j \\
&= \frac{1}{n}\sum_{\mathbf{x}} \frac{\sigma'(a)x_j}{\sigma(a)(1-\sigma(a)}(\sigma(a) - y)
\end{aligned}
$$

Cross-Entropy

The derivative of the sigmoid is:

$$\sigma'(a) = \frac{d}{da}\left(\frac{1}{1+e^{-a}}\right) = \frac{e^{-a}}{(1+e^{-a})^2} = \sigma(a)(1-\sigma(a))$$

yielding:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_{\mathbf{x}} x_j(\sigma(a)-y).$$

which is a beautiful expression since it shows that the rate of learning depends only on how well the output unit is approximating the desired output! In other terms, *it shows that the larger the error, the faster the unit will learn*.

Cross-Entropy

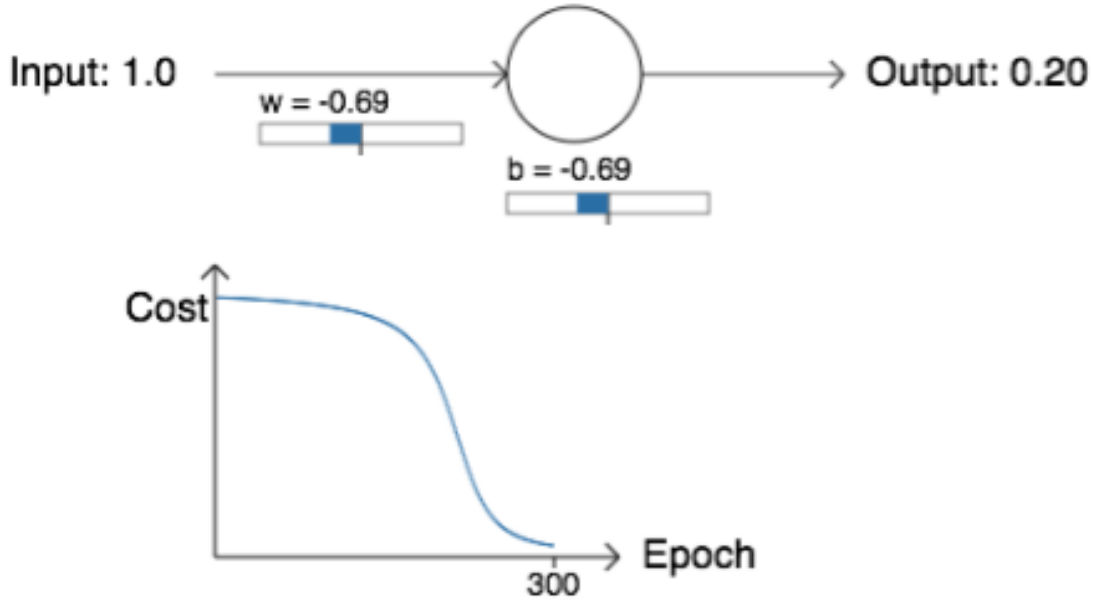
In a similar way, one can show that:

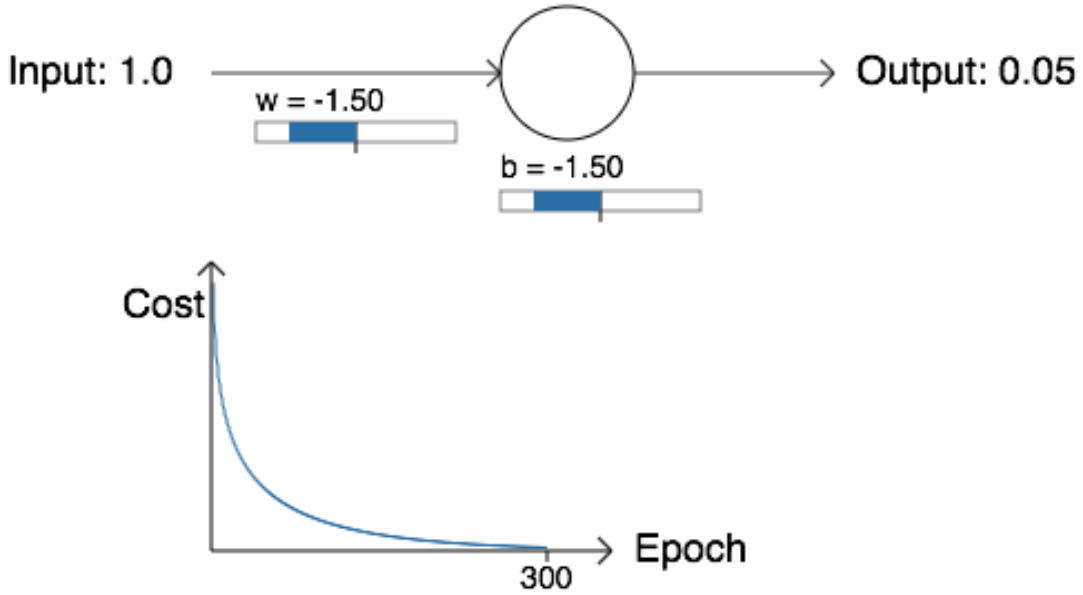
$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_{\mathbf{x}} (\sigma(a) - y).$$

Learning with the Cross-Entropy

first graph: the graph of the **quadratic** cost function when the weights are initialized to $w = 2.0$ and $b = 2.0$.

second graph: the graph of the **cross-entropy** cost function when the weights are initialized to $w = 2.0$ and $b = 2.0$.





Cross-Entropy

All the discussion so far focused on a single neuron. However, it's easy to generalise the cross-entropy to many-neuron, multi-layer networks. Let us assume that $y = y_1, y_t, \ldots$ are the desired values at the output neurons. For this more general case, the cross-entropy cost function is defined as:

$$C = -\frac{1}{n} \sum_x \sum_j [y_j \ln z_j^L + (1 - y_j) \ln(1 - z_j^L)].$$

Soft-Max + Log-Likelihood

Up to now we considered architectures where output units are based on the sigmoid activation function and showed that, in these cases, cross-entropy is particularly advantageous as a cost function.

Another very popular architecture having the same benefits and particularly fitting when the underlying problem requires the prediction of a distribution of probabilities is based on the soft-max activation unit:

$$z_j^L = \frac{e^{a_j^L}}{\sum_k e^{a_k^L}}$$

and the Log-likelihood cost:

$$C \equiv -\ln z_y^L$$

Soft-Max

As it should be apparent, the soft-max activation functions guarantee that:

- each activation $z_j^L \in (0, 1)$
- $\sum_j z_j^L = 1$

That is, the collective behavior of the output units can be interpreted as predicting a distribution of probabilities where $z_j^L = P(j|x)$.

Log-likelihood

Even though it might not be apparent at first, the log-likelihood function behaves exaclty as one would expect from a cost function:

- if the network compute the correct output, then $z_y^L \approx 1$ and $C = -\ln z_y^L \approx 0$.
- if the network *does not* compute the correct output, then $z_y^L \approx 0$ and C will assume a large value.

Slow Learning Problem

As mentioned the soft-max + log-likelihood architecture does not suffer from the slow learning problem. It is not hard to show, in fact, that:

- $\frac{\partial C}{\partial w^L_{jk}} = z^{L-1}_k (z^L_j - y_j)$
- $\frac{\partial C}{\partial b_j} = z^L_j - y_j$

Note: in the last few slides we have been slightly abusing the notation w.r.t. the symbol y. In previous slides we used it to denote the index of the *correct* label, here we are using it to denote the vector having the component corresponding to the correct classification set to 1 and all others set to 0.

Exercise

- show that $\delta_j^L = \frac{\partial C}{\partial a_j^L} = z_j^L - y_j$
- use this result along with `BP3` and `BP4` to prove the identities in the previous slide.

Parameter Initialization

Note: see C. Bishop, H. Bishop, *Springer*, Deep Learning - Foundations and Concepts, §7.2.5.

Initialization of the network parameters can have a significant impact on the generalization performance and the convergence speed. Unfortunately there is very little theory suggesting how to properly initialize the parameters.

Main approaches involve:

- simmetry breaking
- keeping the variance of the parameters constant

Simmetry breaking

Consider a fully connected neural network. If parameters are initialized equally (a common choice would be zero), all neurons will compute the same function and they all would be updated in unison. To address this problem a common a approach is to **initialize the parameters randomly** using either a uniform distribution in the range $[-\epsilon, +\epsilon]$ or with a gaussian distribution $\mathcal{N}(0, \epsilon^2)$.

Keeping the variance constant

The choice of ϵ is very imporant too. Too big or too small weights will contribute to very large or very small gradients, which will cause **gradient explosion** or **gradient implosion**.

A few attempts have been made to find good initialization values. For instance:

- for ReLU activation units the *He initialization* should make the gradient approximatively equal to 1:

$$\epsilon = \sqrt{\frac{2}{M}}$$

where M is the number of inputs of the neuron being initialized.

He initialization

Assume that each layer l of the network evaluates:

$$a_i^l = \sum_{j=1}^{M} w_{ij} z_j^{l-1},$$

$$z_i^l = \mathrm{ReLU}(a_i^l).$$

Assume that weights are initialized drawing from a Gaussian $\mathcal{N}(0, \epsilon^2)$ and assume that outputs of units at layer $l-1$ have zero mean and variance λ^2. **It can be shown that**:

$$\mathrm{var}[z_j^l] = \frac{M}{2}\epsilon^2\lambda^2.$$

The main idea in He initialization is to **keep the variance constant between layers**, which implies:

$$\epsilon = \sqrt{\frac{2}{M}}.$$

Xavier (a.k.a. Glorot) Initialization

Xavier initialization, inspired by the work of Xavier Glorot, is designed for activation functions symmetric around 0 (e.g., $\mathbf{sigmoid}$ and $\mathbf{tanh}$). It samples initialization weights from $U[-\epsilon, \epsilon]$:

- Xavier initialization: $\epsilon = \frac{1}{\sqrt{M}}$,
- Normalized Xavier initialization: $\epsilon = \frac{6}{\sqrt{N+M}}$.

Here, M is the number of inputs to the neuron, and N is the number of units in the layer.

Convergence

Note: see C. Bishop, H. Bishop, *Springer*, Deep Learning - Foundations and Concepts, §7.3.

A few interesting facts can be proven about the convergence of the gradient descent method.

1. the components of the $\mathbf{w}$ **evolve independently**;
2. gradient descent leads to a **linear convergence** in the neighboroud of a minimum;
3. the **rate of convergence** is governed by $1 - \left(\frac{2\lambda_{\min}}{\lambda_{\max}}\right)$

Let us consider a quadratic approximation of the error function around a point $\mathbf{w}^\star$ that is a minimum of the error function:

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \nabla E(\mathbf{w}^\star)^\top (\mathbf{w} - \mathbf{w}^\star) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^\top \mathbf{H} (\mathbf{w} - \mathbf{w}^\star).$$

In this case there is no linear term because $\nabla E = 0$ at $\mathbf{w}^\star$, yielding:

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^\top \mathbf{H} (\mathbf{w} - \mathbf{w}^\star).$$

Consider now the eigenvalue equation for the Hessian matrix:

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i$$

where $\{\mathbf{u}_i\}_i$ are the eigenvectors of $\mathbf{H}$ and form an orthonormal basis:

$$\forall i : \|\mathbf{u}_i\| = 1 \quad \wedge \quad \forall i, j : \mathbf{u}_i^\top \mathbf{u}_j = \delta_{ij}.$$

Since $\{\mathbf{u}_i\}_i$ is an orthonormal basis, we can rewrite any vector in terms of a linear combination of $\mathbf{u}_i$ vectors, which allow us to write:

$$\mathbf{w} - \mathbf{w}^\star = \sum_i \alpha_i \mathbf{u}_i.$$

$\delta_{ij} = I_{i=j} = \begin{cases} 1, & \text{if i=j;} \\ 0, & \text{otherwise.} \end{cases}$

Given the above discussion, we can show that:

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2.$$

Proof:

$$\begin{aligned}
E(\mathbf{w}) &= E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w}-\mathbf{w}^\star)^\top \mathbf{H}(\mathbf{w}-\mathbf{w}^\star) \\
&= E(\mathbf{w}^\star) + \frac{1}{2}\left(\sum_i \alpha_i \mathbf{u}_i\right)^\top \mathbf{H}\left(\sum_i \alpha_i \mathbf{u}_i\right) \\
&= E(\mathbf{w}^\star) + \frac{1}{2}\left(\sum_i \alpha_i \mathbf{u}_i\right)^\top \left(\sum_i \alpha_i \mathbf{H}\mathbf{u}_i\right) \\
&= E(\mathbf{w}^\star) + \frac{1}{2}\left(\sum_i \alpha_i \mathbf{u}_i\right) \left(\sum_i \alpha_i \lambda_i \mathbf{u}_i\right) \\
&= E(\mathbf{w}^\star) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2
\end{aligned}$$

Implying that $\nabla E = \sum_i \alpha_i \lambda_i \mathbf{u}_i$:

Proof:

$$\nabla E(\mathbf{w}) = \nabla \left(E(\mathbf{w}^\star) + \frac{1}{2} \sum_i \lambda_i \alpha_i^2 \right)$$
$$= \frac{1}{2} \sum_i \lambda_i 2 \alpha_i \nabla \alpha_i$$

Let's now compute $\nabla \alpha_i$, let's start from recalling that $\mathbf{w} - \mathbf{w}^\star = \sum_j \alpha_j \mathbf{u}_j$, yielding:

$$\mathbf{u}_i^\top (\mathbf{w} - \mathbf{w}^\star) = \mathbf{u}_i^\top \left(\sum_j \alpha_j \mathbf{u}_j \right)$$

$$\mathbf{u}_i^\top (\mathbf{w} - \mathbf{w}^\star) = \alpha_i$$

$$\sum_j w_j u_{i_j} - \sum_j w_j^\star u_{i_j} = \alpha_i$$

$$\frac{\partial}{\partial w_k} \left(\sum_j w_j u_{i_j} - \sum_j w_j^\star u_{i_j} \right) = u_{i_k} = \frac{\partial \alpha_i}{\partial w_k} \Rightarrow \nabla \alpha_i = \mathbf{u}_i.$$

Summarising, we have:

- $\mathbf{w} - \mathbf{w}^\star = \sum_i \alpha_i \mathbf{u}_i, \quad \Rightarrow \quad \Delta\mathbf{w} = \sum_i \Delta\alpha_i \mathbf{u}_i,$
- $\nabla E = \sum_i \alpha_i \lambda_i \mathbf{u}_i,$
- $\Delta\mathbf{w} = -\eta \nabla E.$

These relations allow us to conclude that:

$$\Delta\alpha_i = -\eta\lambda_i\alpha_i \quad \Rightarrow \quad \alpha_i^{\text{new}} = (1 - \eta\lambda_i)\alpha_i^{\text{old}}$$

Proof:

$$\Delta\mathbf{w} = \sum_i \Delta\alpha_i \mathbf{u}_i \quad \Rightarrow \quad -\eta\nabla E = \sum_i \Delta\alpha_i \mathbf{u}_i \quad \Rightarrow \quad -\eta \sum_i \alpha_i \lambda_i \mathbf{u}_i = \sum_i \Delta\alpha_i \mathbf{u}_i$$

$$\Rightarrow \sum_i \Delta\alpha_i \mathbf{u}_i = \sum_i -\eta\lambda_i\alpha_i \mathbf{u}_i$$

Convergence

From $\mathbf{u}_i^\top(\mathbf{w} - \mathbf{w}^\star) = \alpha_i$ it follows that we can intepret the α_i as the distance from the minimum along the $\mathbf{u}_i$ direction.

Also, from $\alpha_i^{\text{new}} = (1 - \eta\lambda_i)\alpha_i^{\text{old}}$, we know that these **distances evolve independently** and at each step it is reduced by a quantity proportional to $(1 - \eta\lambda_i)$. After T steps, we have:

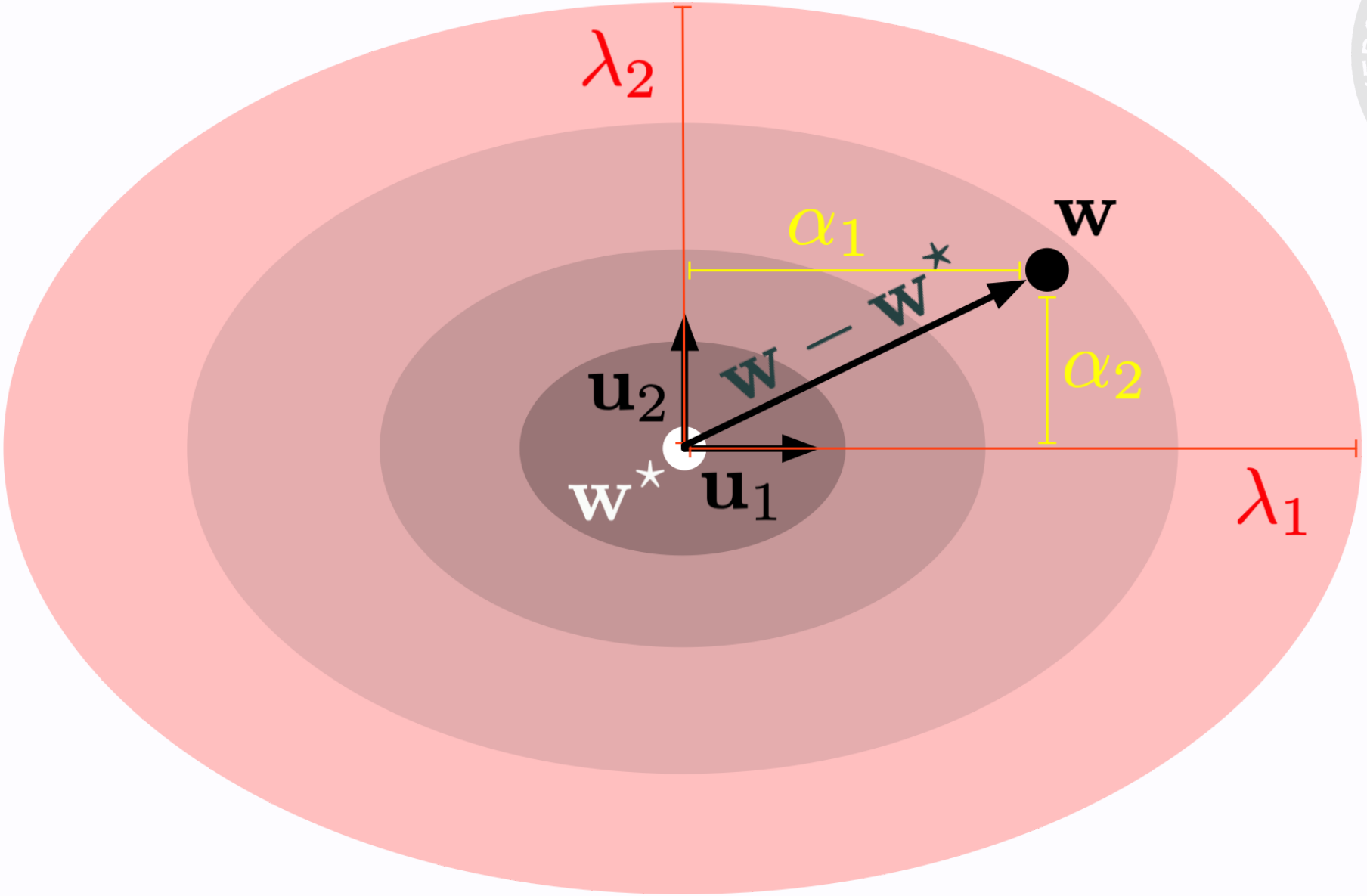
$$\alpha_i^{(T)} = (1 - \eta\lambda_i)^T \alpha_i^{(0)}.$$

Provided that $|1 - \eta\lambda_i| < 1$, the limit as $T \to \infty$ leads to $\alpha_i = 0$, which implies that we reached the minimum of the error function.

Convergence

The **order of convergence is linear** with rate $1 - \eta\lambda_i$ since:

$$\lim_{T\to\infty} \frac{\alpha_i^{(T)} - 0}{(\alpha_i^{T-1} - 0)^1} = 1 - \eta\lambda_i.$$


λ_2
α_1
$\mathbf{w}$
$\mathbf{w} - \mathbf{w}^\star$
α_2
$\mathbf{u}_2$
$\mathbf{w}^\star$
$\mathbf{u}_1$
λ_1

By increasing η we can improve the speed of the convergence, but we must ensure that $|1 - \eta\lambda_i| < 1$, implying that **to ensure convergence** we must set $\eta < 2/\lambda_{\max}$.

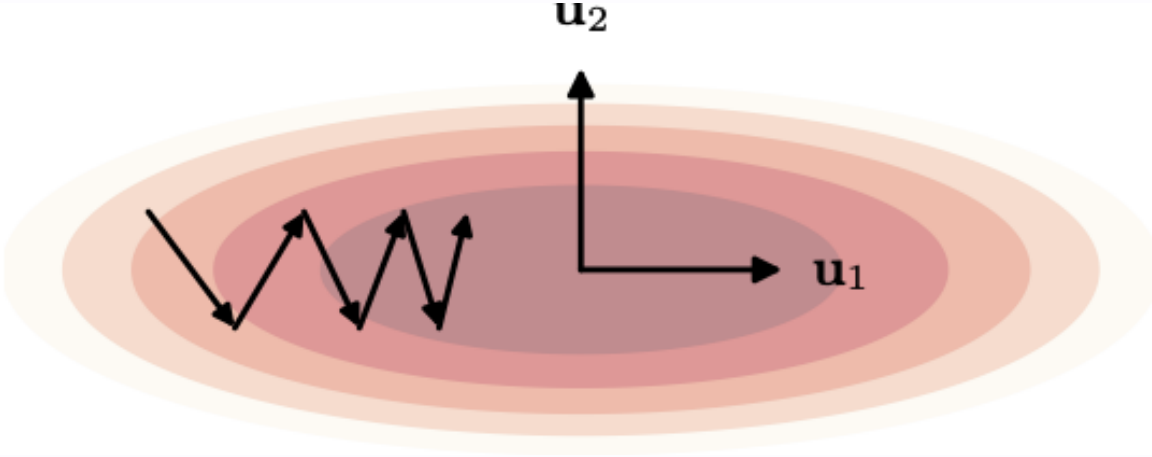
The **fastest convergence** is obtained when $\eta = 1/\lambda_{\max}$. In this case the rate of convergence in the direction of $\lambda_{\max}$ is 0, which implies we will reach the minimum in a single step.

Assuming to set $\eta = 1/\lambda_{\max}$, the direction in which we converge the slowest is $\lambda_{\min}$. In this case the rate of convergence is $1 - \frac{\lambda_{\min}}{\lambda_{\max}}$.

This implies that the worst case scenario is when $\lambda_{\min}$ is small compared to $\lambda_{\max}$, in which case the rate of convergence is approximately 1 and the convergence will be very slow.

The **reciprocal** of $\lambda_{\min}/\lambda_{\max}$ is the **condition number** of the Hessian matrix. The larger the condition number, the slower will be the convergence of gradient descent.

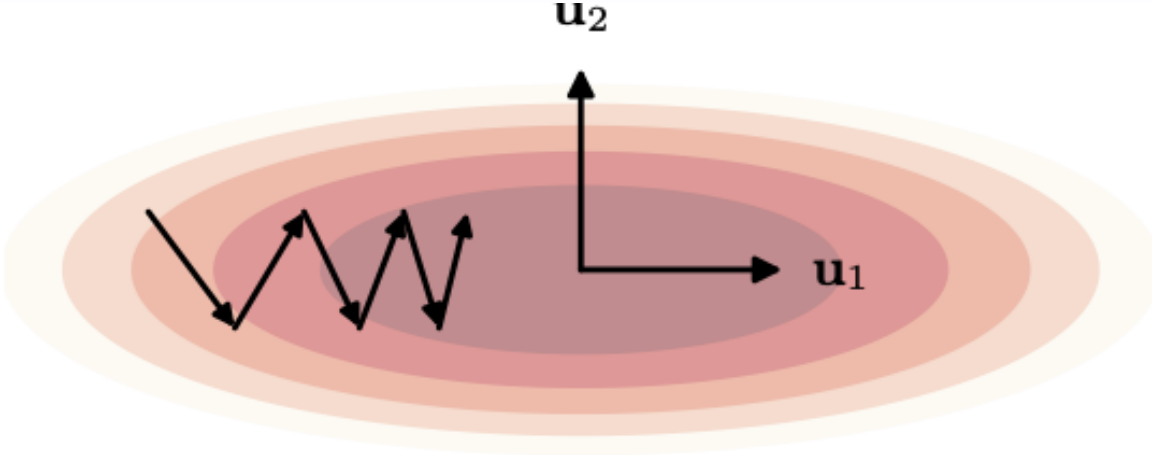
Notice that when $\lambda_{\min} \ll \lambda_{\max}$, the error function will have a very elongated shape, which will make the convergence very slow since the gradient will be very small in the direction of $\lambda_{\min}$.



Momentum & Learning Rate Scheduling

note: see C. Bishop, H. Bishop, *Springer*, Deep Learning - Foundations and Concepts, §7.3.1 and §7.3.2.

When convergence is slow, it can be helpful to add a momentum term to the gradient descent formula. This adds inertia to the motion and smooths out the obscillations shown on the right.



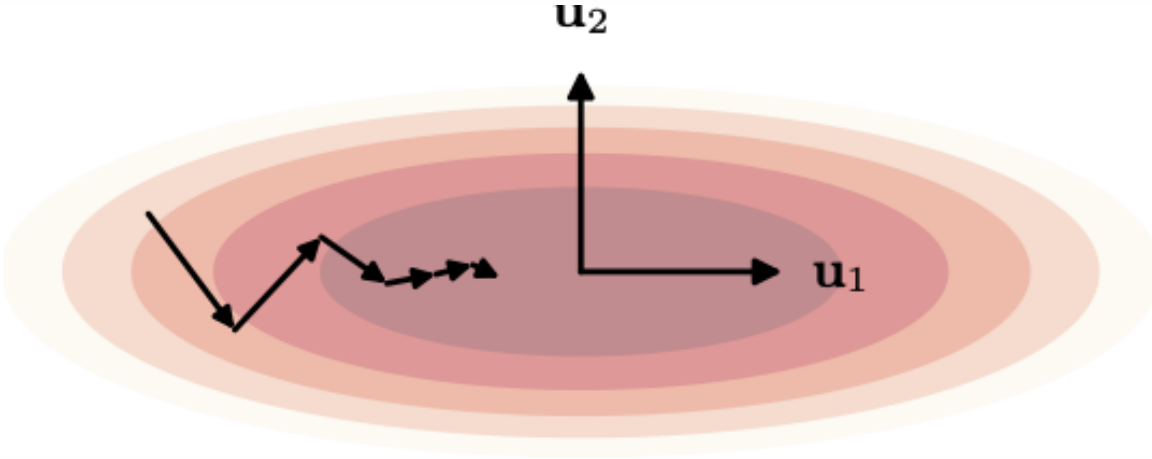
Let us recall that the update formula for the network weights is:

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} + \Delta\mathbf{w}^{(\tau-1)}$$

When we add the momentum, the term $\Delta\mathbf{w}^{(\tau-1)}$ becomes:

$$\Delta\mathbf{w}^{(\tau-1)} = -\eta\nabla E(\mathbf{w}^{(\tau-1)}) + \mu\Delta\mathbf{w}^{(\tau-2)}$$

where μ is the **momentum parameter**.



In a region of low curvature, if we make the approximation that the gradient and the momentum are not changing, then after a long series of updates we have:

$$\begin{aligned}\Delta \mathbf{w} &= -\eta \nabla E + \mu[-\eta \nabla E + \mu[-\eta \nabla E + \ldots]] \\ &= -\eta \nabla E\{1 + \mu + \mu^2 + \ldots\} = -\frac{\eta}{1-\mu}\nabla E,\end{aligned}$$

which, provided that $\mu < 1$, implies that the momentum is increasing the learning rate by a factor of $\frac{1}{1-\mu}$.

In a region of high curvature, where gradient descent is oscillatory, successive contributions of the momentum tend to cancel and the effective learning rate will be close to η.

A tipical value for the momentum parameter is 0.9.

Learning Rate Scheduling

It is advantageous to change the learning rate η during learning. In practice, the best results are obtained using a larger value for η at the start of training and then reducing the learning rate over time:

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta^{(\tau-1)} \nabla E(\mathbf{w}^{(\tau-1)})$$

Examples of learning schedules:

- **linear**: $\eta^{(\tau)} = (1 - \tau/K)\eta_0 + (\tau/K)\eta_K$;
  where η_0, η_K, K are three parameters and the formula is applied up to K steps. After these K steps, η is kept fixed at η_K;
- **power law**: $\eta^{(\tau)} = \eta_0(1 + \tau/s)^c$;
- **exponential decai**: $\eta^{(\tau)} = \eta_0 c^{\tau/s}$

Further algorithms

Additional refinements can be obtained observing that the change in the learning rate can be optimized separately for each direction in parameter space, i.e. for each w_i. Based on this ideas the three most popular schedule methods are:

- **AdaGrad** (short for *Adaptive Gradient*): reduce each learning parameter over time by using the accumulated sum of squares of all derivatives calculated for that parameter; the idea is to reduce the learning rate for parameters that have received large updates in the past;
- **RMSProp** (short for *root mean square propagation*): replace the sum of squared gradients of AdaGrad with an exponentially weighted average. Fixes a problem with AdaGrad where the weight updates tend to become too small.
- **Adam** (short for *Adaptive moments*): combines RMSProp with momentum. It is probably the most used optimization method in deep learning.

Normalization

Note: see C. Bishop, H. Bishop, *Springer*, Deep Learning - Foundations and Concepts, §7.4.

Coping with values that vary in very different ranges as well as having to deal with **vanishing** and **exploding** gradients are main problems in deep learning.

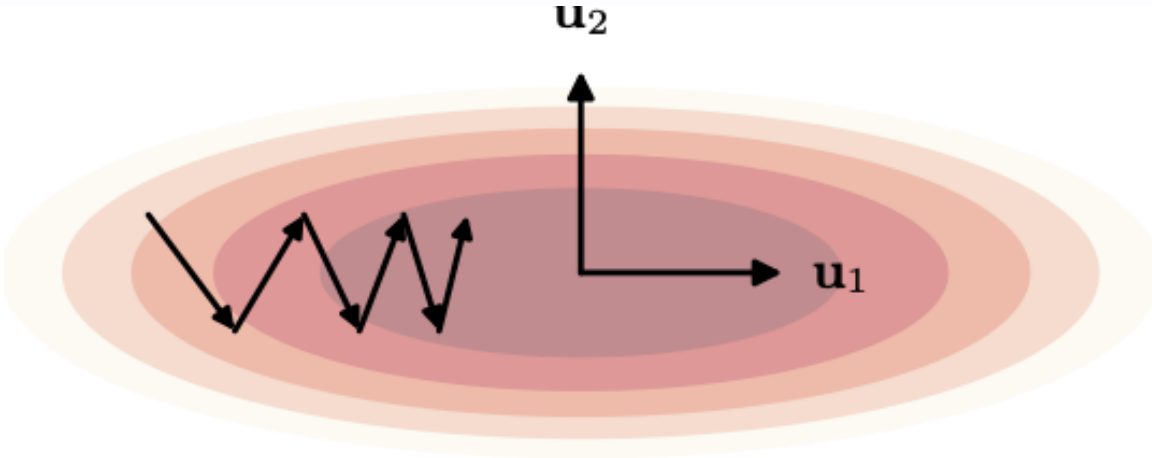
Weight normalization try to deal with these problems by keeping the values computed by the network in a reasonable range.

Three main approaches:

- data normalization;
- batch normalization;
- layer normalization.

Data Normalization

If the data set has input variables that span very different ranges, then a change in one dimension will produce a much larger change in the output w.r.t. a change in another dimension.



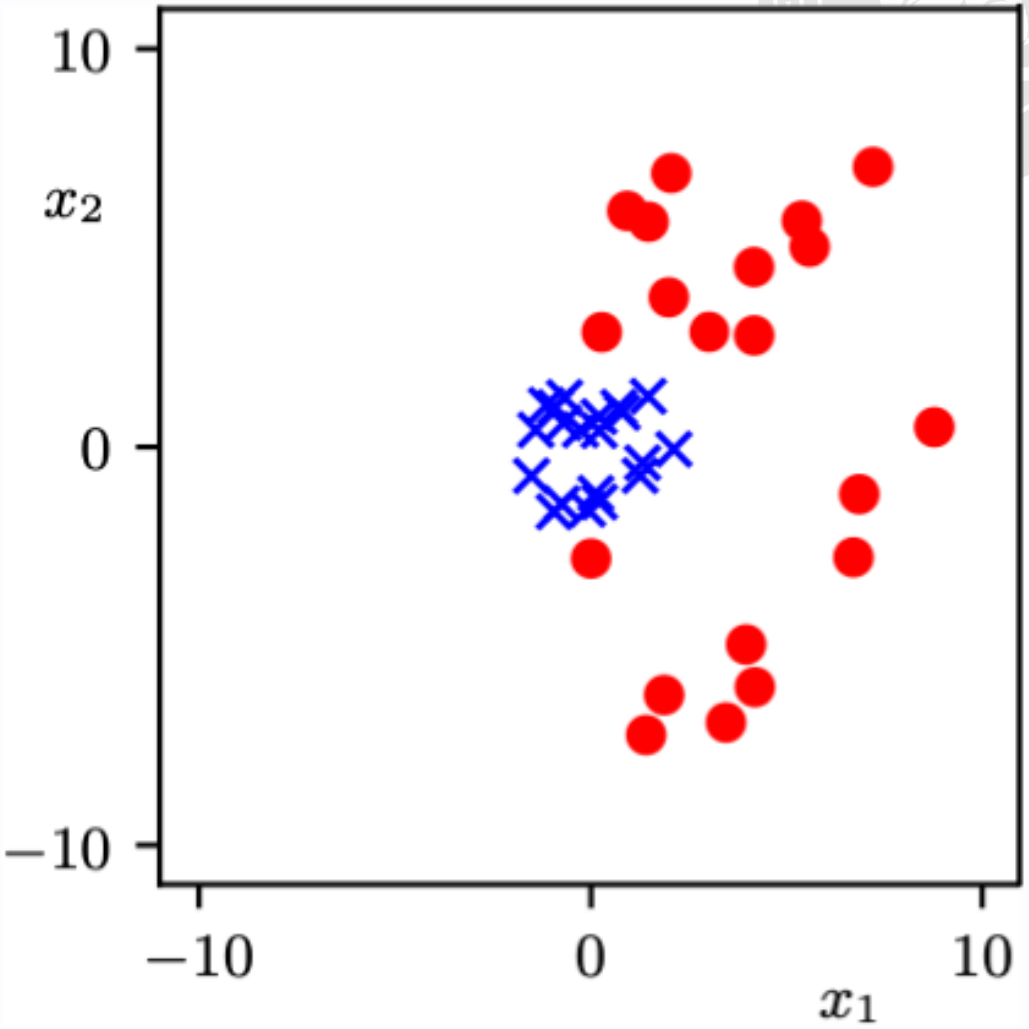
To cope with the problem, we first compute mean and variance for each dimension i:

$$\mu_i = \frac{1}{N}\sum_{n=1}^{N} x_{ni},$$

$$\sigma_i^2 = \frac{1}{N}\sum_{n=1}^{N}(x_{ni} - \mu_i)^2;$$

then we rescale all data points using:

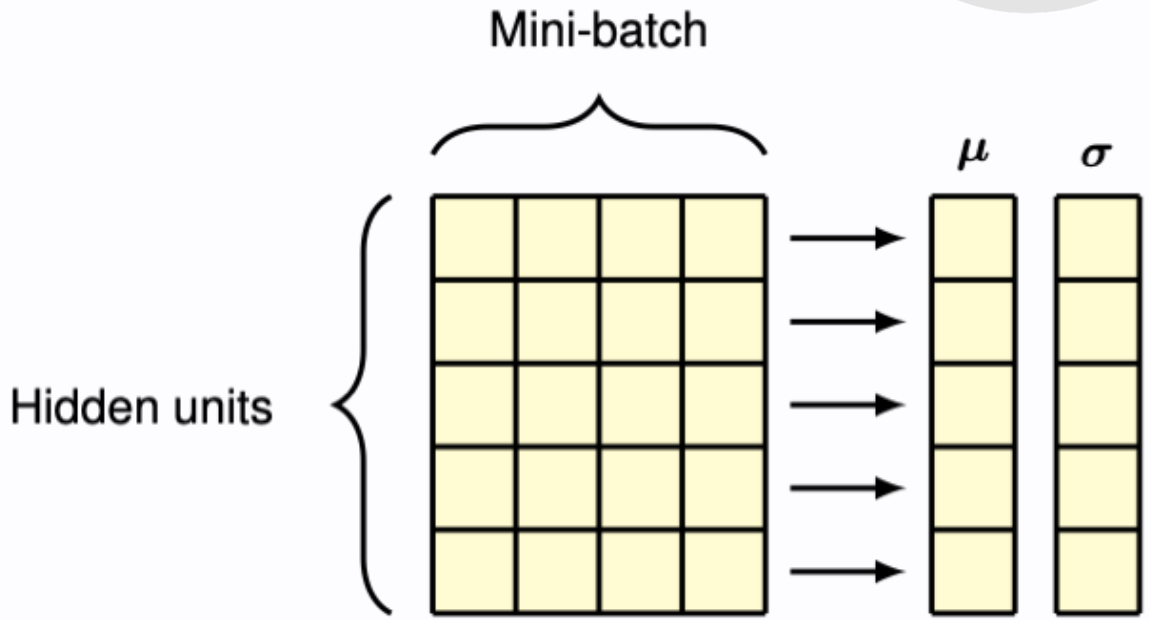
$$\tilde{x}_{ni} = \frac{x_{ni} - \mu_i}{\sigma_i}$$



Batch normalization

The same reasoning can be applied to the variables (weights) at each hidden layer. Unfortunately, **normalization for those values cannot be done once for all**, the computation need to be performed every time the variables are updated.

Batch normalization works by normalizing, across the examples of the mini-batch the values computed at each layer of the network by each unit i.



Let us consider an hidden unit $z_i = h(a_i)$, the interesting quantities are the pre-activation values a_i and the post-activation values z_i. One can normalize the pre or the post activations, both approaches work well in practice.

For instance, to normalize the pre-activation values one would compute:

$$\mu_i = \frac{1}{K} \sum_{n=1}^{K} a_{ni}$$

$$\sigma_i^2 = \frac{1}{K} \sum_{n=1}^{K} (a_{ni} - \mu_i)^2$$

$$\hat{a}_{ni} = \frac{a_{ni} - \mu_i}{\sqrt{\sigma_i^2 + \delta}}$$

where K is the size of the mini-batch and δ avoids numerical issues when σ_i^2 is small.

This kind of normalization reduce the representational capability of the hidden units. To compensate for this, one can **rescale the pre-activation values** to have mean β_i and standard deviation γ_i:

$$\tilde{a}_{ni} = \gamma_i \hat{a}_{ni} + \beta_i.$$

While originally the mean and variance across a minibatch were computed by a complex function of all weights and biases, now they are determined by two simple independent parameters, which turn out to be much easier to learn by gradient descent.

Once training completes, we will not have mini-batches to compute the normalization factors. To cope with that a moving average of those factors is kept during the training and used at **inference time**.

$$\bar{\mu}_i^{(\tau)} = \alpha \bar{\mu}_i^{(\tau-1)} + (1-\alpha)\mu_i$$
$$\bar{\sigma}_i^{(\tau)} = \alpha \bar{\sigma}_i^{(\tau-1)} + (1-\alpha)\sigma_i$$

where $0 \leq \alpha \leq 1$.

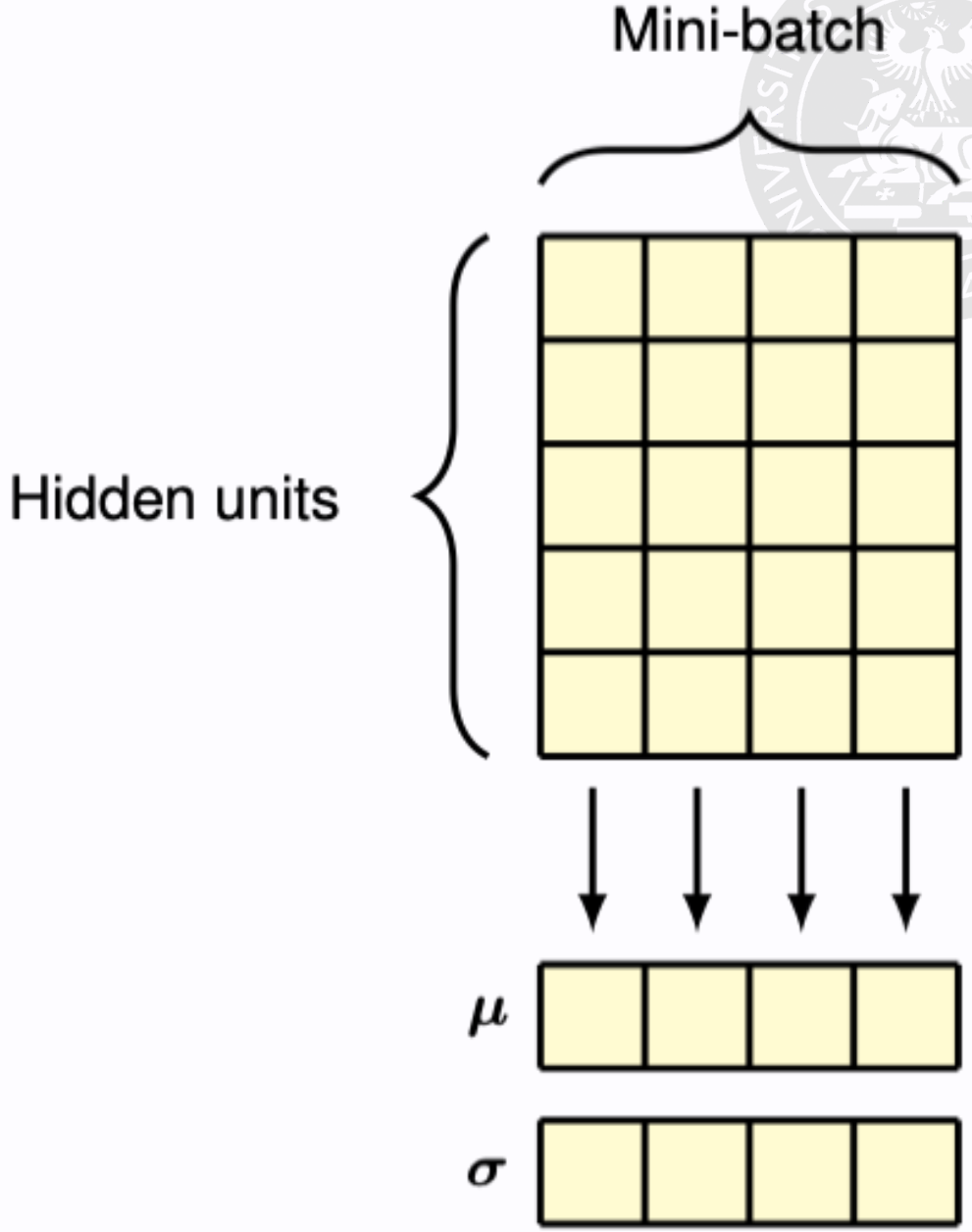
Batch normalization proves to be very effective in practice, but it is not totally clear why.

It was originally motivated as a way to counter the **internal covariate shift** problem: occurring when updates to weights in the earlier layers of the network change the distribution values seen by later layers.

Later studies have shown that covariate shift is not a significant factor and that the improvements stems from **smoothing the error function landscape**.

Layer normalization

Instead of normalizing across examples within a mini-batch for each hidden unit separately, layer normalization **normalizes across the hidden unit values for each data point separately**.



Layer normalization updates the pre-activation values as it follows:

$$\mu_n = \frac{1}{M}\sum_{i=1}^{M} a_{ni}$$

$$\sigma_n^2 = \frac{1}{M}\sum_{i=1}^{M}(a_{ni} - \mu_i)^2$$

$$\hat{a}_{ni} = \frac{a_{ni} - \mu_n}{\sqrt{\sigma_n^2 + \delta}}$$

where n ranges over the examples and M is the number of hidden units in the layer.

Also in this case, **additional learnable parameters** β_i and λ_i are introduced (using the same ideas as for batch normalization).

Note: in this case there is no need to keep moving averages to normalize data at inference time.