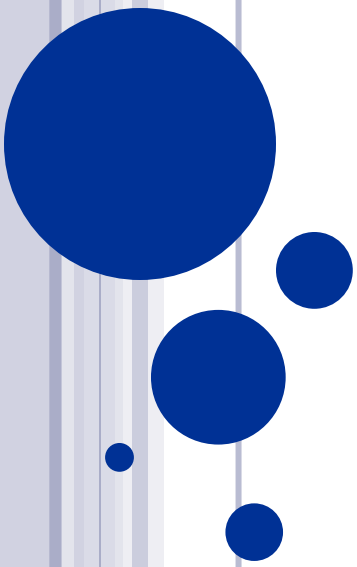


Deep Learning

Elisabetta Fersini

elisabetta.fersini@disco.unimib.it

Hyper-parameter tuning

- Hyper-paramters are variables that need to be set and have a strong impact to the Deep Learning generalization abilities
  - Hyper-parameters for the training algorithm
  - Hyper-parameters for the neural network model

- Choosing hyper-parameter values is formally equivalent to the question of model selection:
  - given a family or set of learning algorithms, how to pick the most appropriate one inside the set?

Hyper-parameter tuning

- A hyper- parameter for a training algorithm A is a variable to be set prior to the actual application of A to the data
  - one that is not directly selected by the learning algorithm itself.

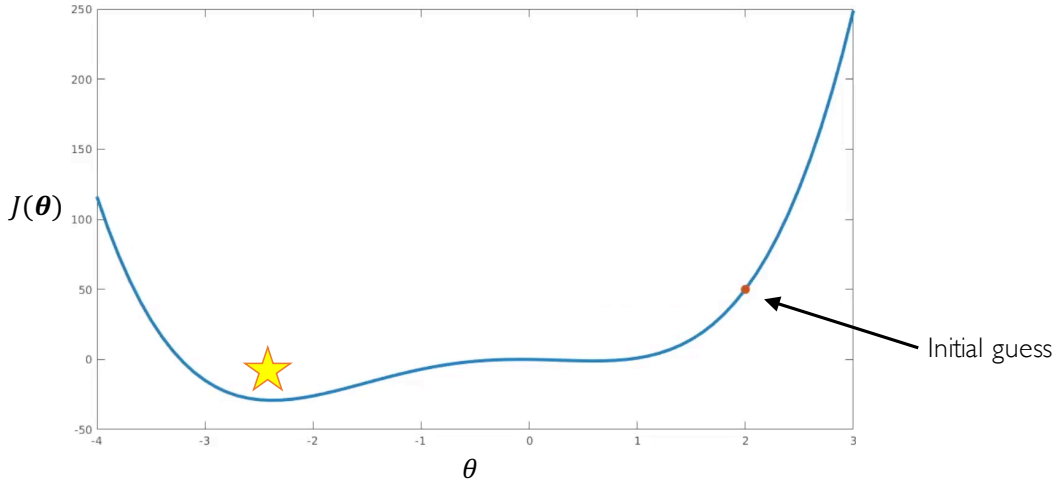
- Hyper-parameters can be fixed by hand or tuned by an algorithm, but their value has to be fixed.
  - The value of some hyper-parameters can be selected based on the performance of A on its training data, but most cannot.
  - For any hyper-parameter that has an impact on the effective capacity of a learner, we should use out-of-sample data
    - validation set performance, cross-validation error.

HYPER-PARAMETER TUNING

- Loss Functions Can Be Difficult to Optimize by SGD
  - **Remember:** Optimization through gradient descent

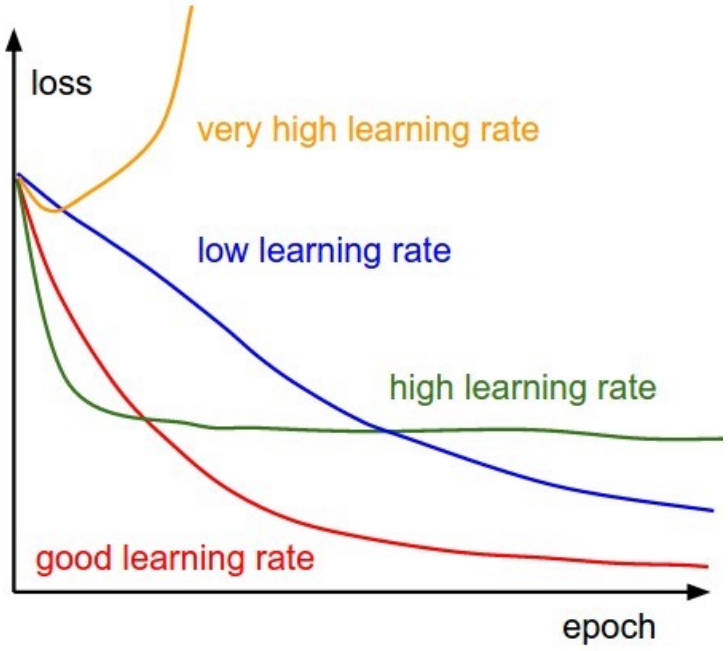
$$\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$$

  - How can we set the LEARNING RATE?
    - This is one of the most important hyper-paramter that we have to take care!



Hyper-parameter tuning

- Learning Rate:
  - *Small learning rate* converges slowly and gets stuck in false local minima
  - *Large learning rates* overshoot, become unstable and diverge
  - *Stable learning rates* converge smoothly and avoid local minima



HYPER-PARAMETER TUNING

- Learning Rate:
  - *Small learning rate* converges slowly and gets stuck in false local minima
  - *Large learning rates* overshoot, become unstable and diverge
  - *Stable learning rates* converge smoothly and avoid local minima

- In mini-batch SGD with standardized input the learning rate is usually less than 1 and greater than 10^{-6}
  - A typical choice is 0.001
  - If you have the chance to tune this hyper-parameter, you should take care of it

- How to deal with this choice?
  - OPTION 1: Try lots of different learning rates and see what works "just right"
  - OPTION 2: Design an adaptive learning rate that "adapts" to the landscape

Hyper-parameter tuning

- In mini-batch SGD with standardized input the learning rate is usually less than 1 and greater than 10^{-6}
  - A typical choice is 0.001
  - If you have the chance to tune this hyper-parameter, you should take care of it

- How to deal with this choice?
  - OPTION 1: Try lots of different learning rates and see what works "just right"
  - OPTION 2: Design an adaptive learning algorithm that "adapts" to the landscape

READING

Bengio, Yoshua. "Practical recommendations for gradient-based training of deep architectures." Neural networks: Tricks of the trade. Springer, Berlin, Heidelberg, 2012. 437-478.

HYPER-PARAMETER TUNING

- Learning rate schedule:
  - Learning rates are no longer fixed
  - Can be made larger or smaller depending on:
    - how large gradient is
    - how fast learning is happening
    - size of particular weights
    - etc...

- One of the choices is to use learning rate decay w.r.t to a time constant:

$$\eta_t = \frac{\eta_0 \tau}{\max(t, \tau)}$$

  - Which keeps the learning rate constant for the first τ steps and then decreases it
  - for $\tau \to \infty$ the learning rate is constant over the training iterations

HYPER-PARAMETER TUNING

- An alternative choice is to use brute force adaptive learning rate heuristics:
  - at regular intervals during training
  - using a fixed small subset of the training set
  - continue training with N different choices of learning rate (all in parallel)

keep the value that gave the best results until the next re-estimation of the optimal learning rate.

- These schedules, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics.

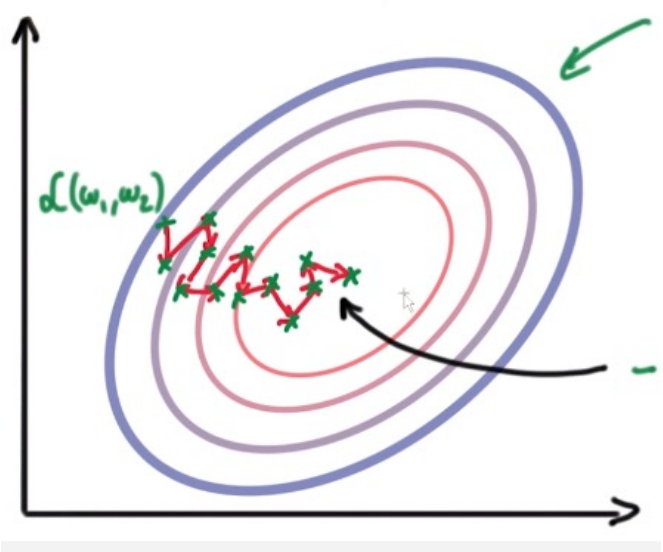
- Additionally, the same learning rate applies to all parameter updates.

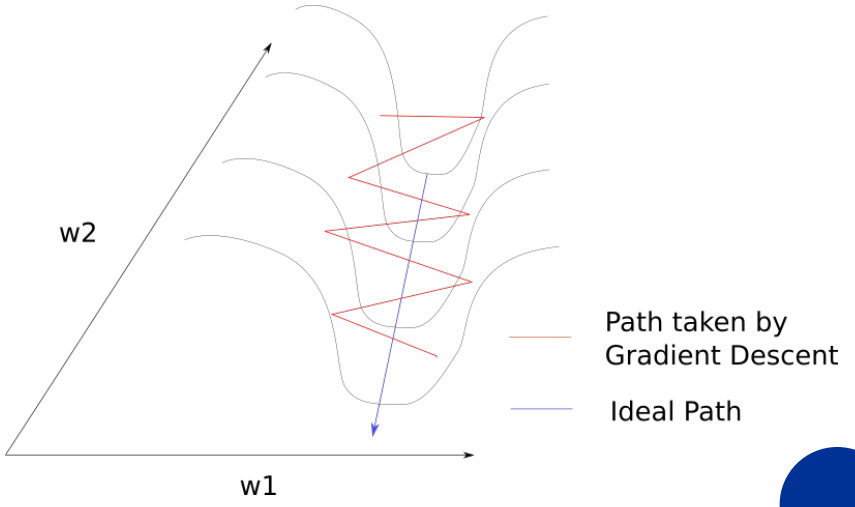
"ADVANCED TRAINING"

- Alternative solution:
  - More advanced training approaches
    - Momentum
    - Nesterov Accelerated Gradient
    - Adagrad
    - Adadelta
    - Adam

Gradient descent optimization algorithms

- SGD suffers where the surface curves much more steeply in one dimension than in another, which are common around local optima.
- SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum.





GRADIENT DESCENT OPTIMIZATION ALGORITHMS

- To overcome this, we introduce MOMENTUM:
  - It takes knowledge from previous steps about where we should be heading.
  - We are introducing a new hyperparameter γ

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

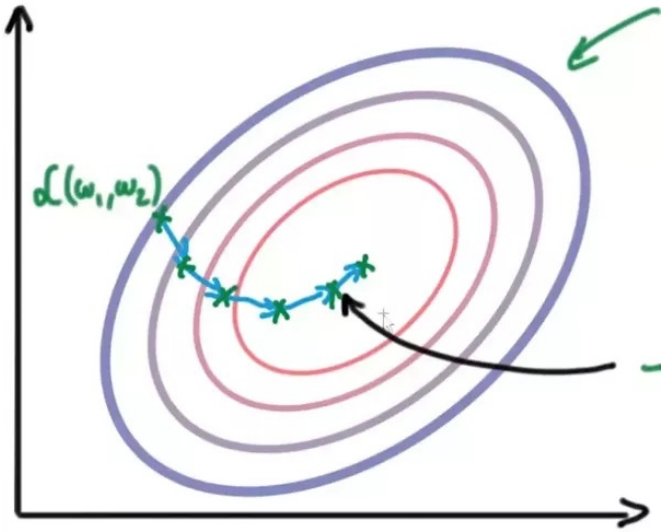
momentum term

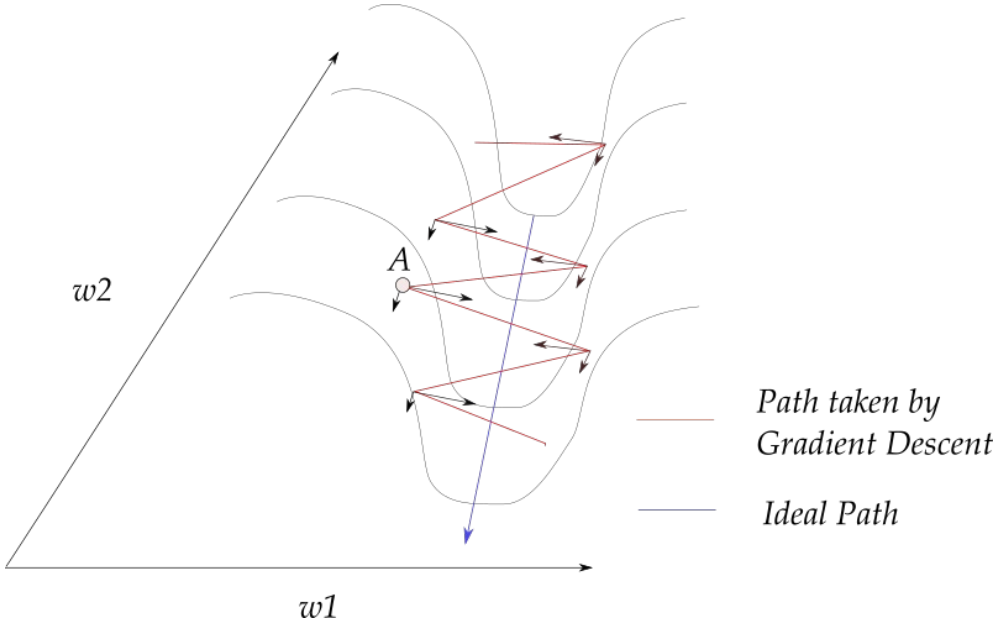
It adds a fraction γ of the update vector of the past time step to the current update vector

- MOMENTUM accelerate SGD in the relevant direction and reduces oscillations

GRADIENT DESCENT OPTIMIZATION ALGORITHMS

- MOMENTUM:





However, MOMENTUM follows the slope…

Gradient descent optimization algorithms

- Nesterov Accelerated Gradient (NAG)
- Differently from the standard momentum approach, it evaluates the gradient after the velocity is applied.
- Basically, we know that we will use our momentum term γv_{t-1} to move the parameters θ.
  - Computing $\theta - \gamma_{vt-1}$ thus gives us an approximation of the next position of the parameters, i.e. a rough idea where our parameters are going to be.
  - We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

GRADIENT DESCENT OPTIMIZATION ALGORITHMS

- Nesterov Accelerated Gradient (NAG)
  - Momentum first computes the current gradient (small blue vector) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector)
  - NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (green vector).

$$v_t = \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

GRADIENT DESCENT OPTIMIZATION ALGORITHMS

- Adagrad
- It adapts the learning rate w.r.t the parameters:
  - It performs smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features
  - It performs larger updates (i.e. high learning rates) for parameters associated with infrequent features.
  - it is well-suited for dealing with sparse data

  - Basically it uses a different learning rate for every parameter θ_i at every time step t

GRADIENT DESCENT OPTIMIZATION ALGORITHMS

- Let $g_{t,i} = \nabla_\theta J(\theta_{t,i})$. be the partial derivative of the objective function w.r.t. to the parameter θ_i at time step t
- The SGD update for every parameter θ_i at each time step t as follows:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

- Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

historical gradient information

G_t is a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. θ_i up to time step t

Gradient descent optimization algorithms

- Adagrad's main weakness is its accumulation of the squared gradients in the denominator:
  - Since every added term is positive, the accumulated sum keeps growing during training.
  - This causes the learning rate to shrink and eventually become infinitesimally small,
    - the algorithm is no longer able to acquire additional knowledge.

learning rate converges to zero with increase of time!

Gradient descent optimization algorithms

- Adadelta
  - Instead of storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients.

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

squared parameter updates

decaying average over past squared gradients

  - It replaces the learning rate η with the root mean squared error of parameter updates

GRADIENT DESCENT OPTIMIZATION ALGORITHMS

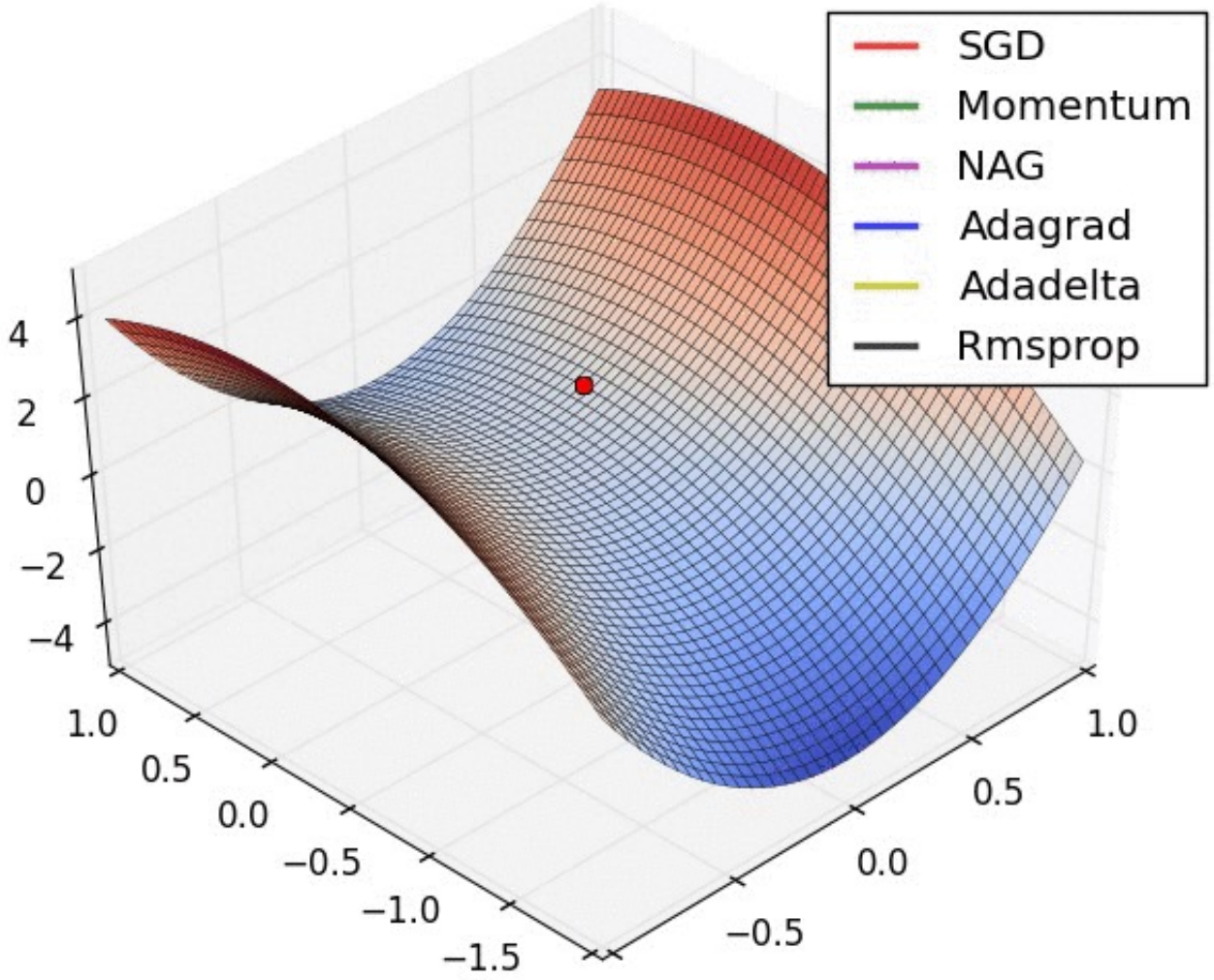
- Adaptive Moment Estimation (Adam)
  - In addition to storing an exponentially decaying average of past squared gradients v_t (like Adadelta), Adam also keeps an exponentially decaying average of past gradients m_t (similar to momentum)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Gradient descent optimization algorithms



READINGS

[ADAMAX] *Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13.*

[NADAM] *Dozat, T. (2016). Incorporating Nesterov Momentum into Adam. ICLR Workshop, (1), 2013–2016.*

[AMSGrad] *Reddi, Sashank J., Satyen Kale, and Sanjiv Kumar. "On the convergence of adam and beyond." ICLR 2018.*

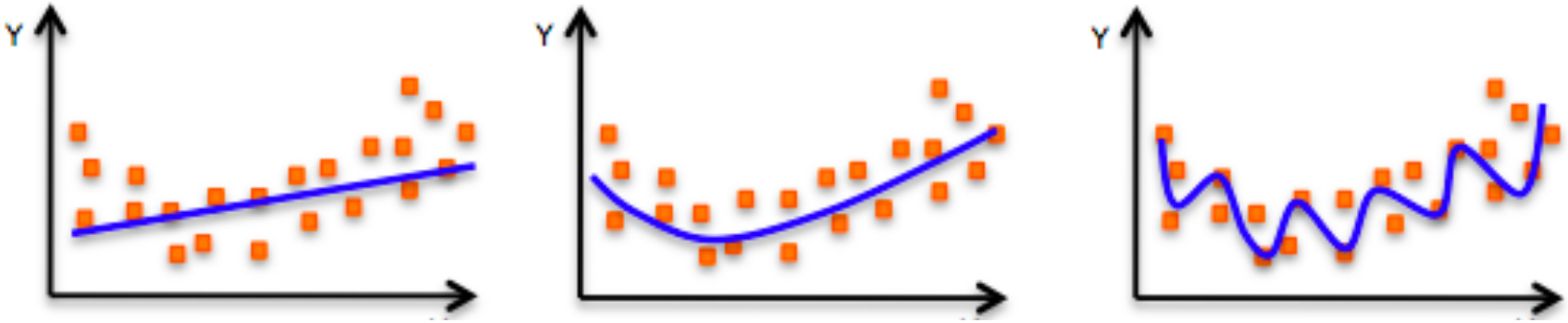
[AdamW] *Loshchilov, I., & Hutter, F. (2019). Decoupled Weight Decay Regularization. In Proceedings of ICLR 2019.*

[QHAdam] *Ma, J., & Yarats, D. (2019). Quasi-hyperbolic momentum and Adam for deep learning. In Proceedings of ICLR 2019.*

[AggMo] *Lucas, J., Sun, S., Zemel, R., & Grosse, R. (2019). Aggregated Momentum: Stability Through Passive Damping. In Proceedings of ICLR 2019.*

AVOID OVERFITTING

- Hyper-parameter settings has impact on the overfitting problem



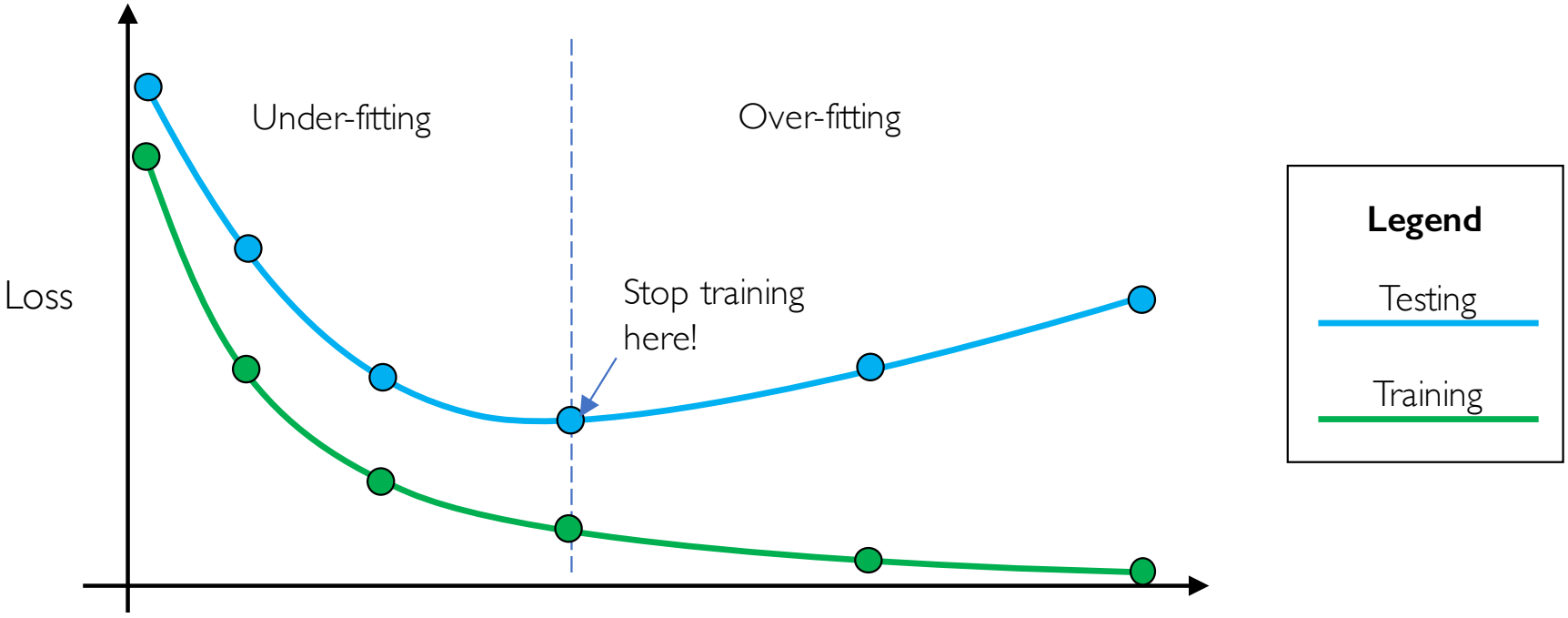
Underfitting
Model does not have capacity to fully learn the data

Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

AVOID OVERFITTING

- Number of training iterations (measured in mini-batch updates)
- This hyper-parameter is particular in that it can be optimized almost for free using the principle of EARLY STOPPING:
  - Stop training before we have a chance to overfit



AVOID OVERFITTING

- EARLY STOPPING in practice (heuristic):
  - It is based on the idea of patience
  - As training proceeds and new candidate selected points T are observed, the patience parameter is increased
  - If we find a new minimum at t:
    - we save the current best model,
    - we update T ← t
    - we increase our patience up to (t + constant) or (t x constant)

AVOID OVERFITTING

- The minimization of the expected risk can be then approximated by (approximately) minimizing the following empirical risk:

$$\underset{w}{\text{minimize}} \frac{1}{|\mathcal{D}|} \sum_{(x_i,t_i)\in\mathcal{D}} E\big(f_w(x_i), t_i\big) + R(\ldots)$$

- Which are the component that could contribute to the **regularization**?
  - $\mathcal{D}$: The training set
  - f: The selected model family
  - E: The error function
  - R: The regularization term
  - The optimization procedure itself

AVOIDING OVERFITTING

- **Data Transformation**: from training set $\mathcal{D}$ to ia new set $\mathcal{D}_R$

- Transformation with stochastic parameters is a function τ_θ with parameters θ which follow some probability distribution
  - Corruption of inputs by Gaussian noise: generating new samples

$$\tau_\theta(x) = x + \theta, \quad \theta \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}).$$

AVOIDING OVERFITTING

- **Data Tranformation** can be distinguished according to the properties of the used transformation and of the distribution of its parameters
- Stochastic parameters: Allow generation of a larger, possibly infinite, dataset. Various strategies for sampling of θ exist:
  - Random: Draw a random θ from the specified distribution
  - Adaptive: θ is the result of an optimization procedure
    - Constrained optimization: θ found by maximizing error under hard constraints
    - Unconstrained optimization: θ found by maximizing modified error function, using the distribution of θ as weighting
    - Stochastic: θ found by taking a fixed number of samples of θ and using the one yielding the highest error

AVOIDING OVERFITTING

- Regularization via the **network architecture**
- The network architecture is represented by a function $f : (w,x) \rightarrow y$, and together with the set W of all its possible weight configurations defines a set of mappings that this particular architecture can realize: $\{f_w : x \rightarrow y \mid \forall w \in W\}$.
- Possible strategy:
  - Weight sharing: reusing a certain trainable parameter in several parts of the network

AVOIDING OVERFITTING

- Regularization via the **error function**

- An example is Dice Loss which is robust to class imbalance

$$D = \frac{2\sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2}$$

$$\frac{\partial D}{\partial p_j} = 2\left[\frac{g_j\left(\sum_i^N p_i^2 + \sum_i^N g_i^2\right) - 2p_j\left(\sum_i^N p_i g_i\right)}{\left(\sum_i^N p_i^2 + \sum_i^N g_i^2\right)^2}\right]$$

AVOIDING OVERFITTING

- Regularization via the **regularization term R**

- R can depend on:
  - the weights w
  - the network output $y = fw(x)$
  - $\partial y/\partial w$ of the output $y = fw(x)$ w.r.t. the weights w
  - $\partial y/\partial x$ of the output $y = fw(x)$ w.r.t. the input x

AVOID OVERFITTING

- Regularization term constrains our optimization problem to discourage complex models
  - Improve generalization of our model on unseen data

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}}\, \mathcal{L}(\Theta) + \lambda R(\Theta)$$

$$= \underset{\Theta}{\operatorname{argmin}}\, \frac{1}{n} \sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i) + \lambda R(\Theta).$$

AVOID OVERFITTING

- REGULARIZATION:
  - The regularizers ***R*** equate complexity with large weights, and work to keep the parameter values low.
  - **How does the regularizers work?**
    - It measure the norms of the parameter matrices
  - L1 regularization
  - L2 regularization

AVOID OVERFITTING

- L1 regularization (lasso):

$$R_{L_1}(\boldsymbol{W}) = ||\boldsymbol{W}||_1 = \sum_{i,j} |\boldsymbol{W}_{[i,j]}|$$

  - L1 regularization makes sure that parameters that are not really very useful are driven to zero

AVOID OVERFITTING

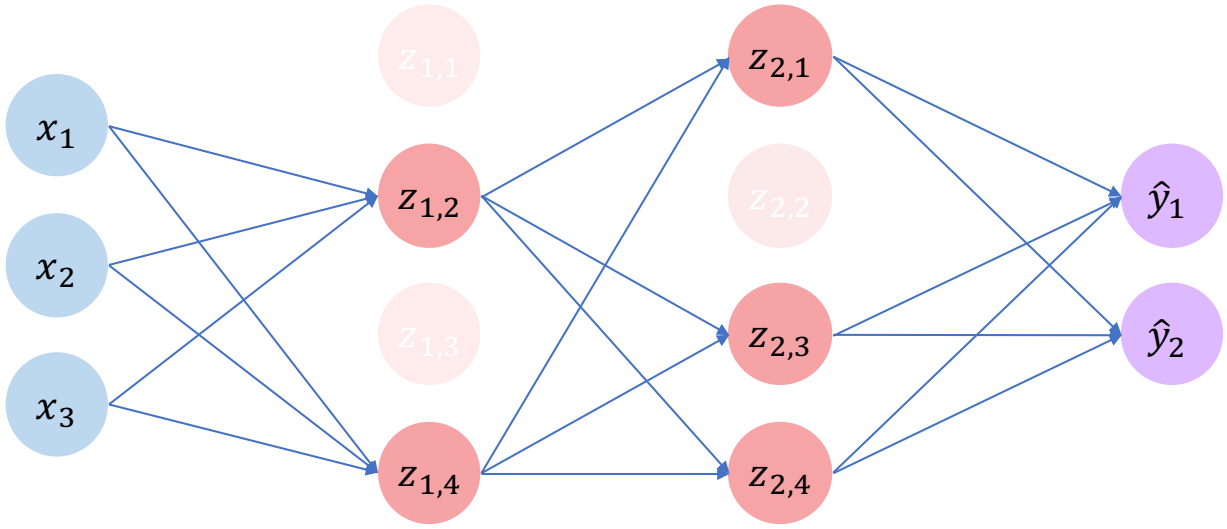
- L2 regularization (weight decay):

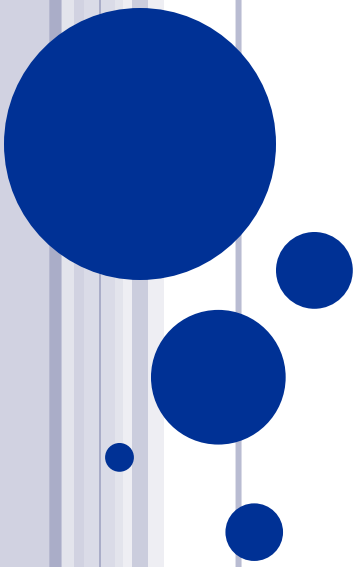
$$R_{L_2}(\boldsymbol{W}) = ||\boldsymbol{W}||_2^2 = \sum_{i,j}(\boldsymbol{W}_{[i,j]})^2$$

It strongly penalizes large values of parameters

AVOID OVERFITTING

- Dropout:
  - During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node





Representation Learning

Representation learning

- There are several motivations behind layer-wise unsupervised pre-training for deep neural networks and therefore representation learning:

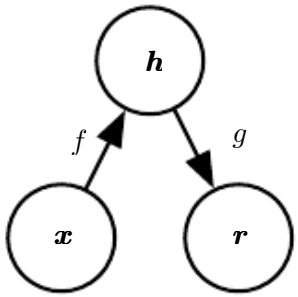
1. Learning features, not just handcrafting them
   - Hand-crafting features is time-consuming and incomplete
2. Most all data is unlabeled, we can make use of it
3. Raw data is sparse (extreme case is one-hot) and we need a more efficient and effective representation
4. We can emulate our brain by learning multiple levels of representation in an unsupervised settings
   - Humans first learn simpler concepts and then compose them to more complex ones

REPRESENTATION LEARNING

- An AUTOENCODER is a neural network that is trained to attempt to copy its input to its output.

- Internally, it has a hidden layer h that describes a code used to represent the input.

- The network may be viewed as consisting of two parts: an encoder function h = f(x) and a decoder that produces a reconstruction r = g(h).



Representation learning

- Copying the input to the output may sound useless, but we are typically not interested in the output of the decoder.

- Instead, we hope that training the autoencoder to perform the input copying task will result in h taking on useful properties.

- One way to obtain useful features from the autoencoder is to constrain h to have smaller dimension than x.
  - The autoencoder is said to be undercomplete

- Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data.

Representation learning

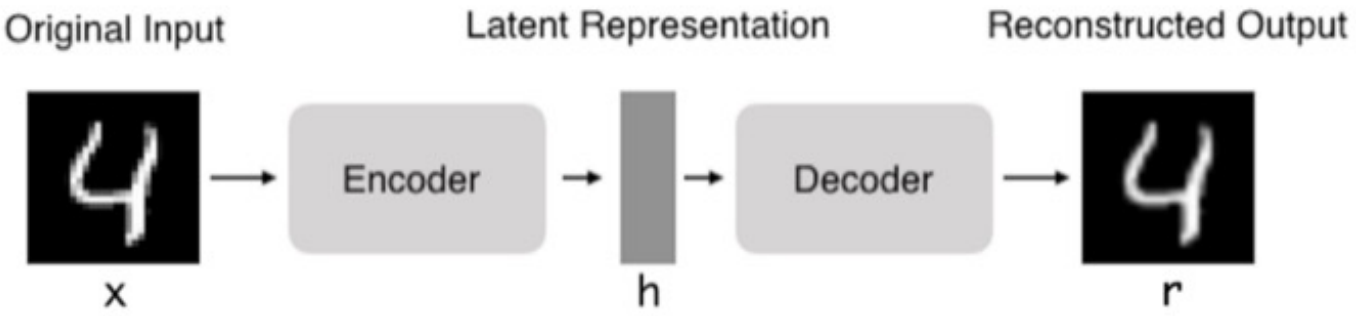
- The learning process is described simply as minimizing a loss function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x})))$$

  - where L is a loss function penalizing g(f(x))) for being dissimilar from x, such as the mean squared error

Representation learning

- Autoencoders
- Deep Autoencoders
- Denoising Autoencoders
- Stacked Denoising Autoencoders



Architecture of an Autoencoder

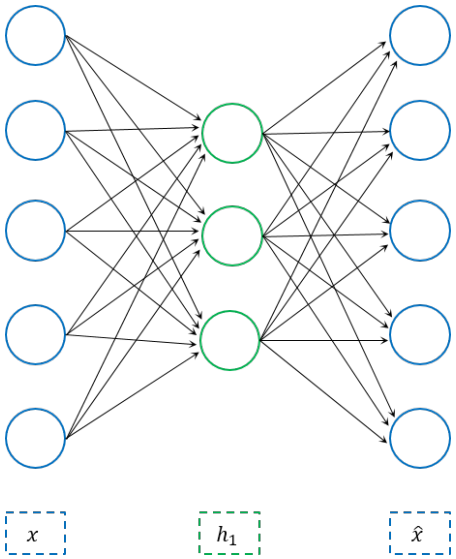
Representation learning

- The simplest Autoencoder has an MLP-like (Multi Layer Perceptron) structure:
  - Input Layer
  - Hidden Layer, and
  - Output Layer

- However, unlike MLP, autoencoders do not require any target data. As the network is trying to learn x itself, the learning algorithm is a special case of unsupervised learning.

Representation learning

- The simplest Autoencoder can then be summarized as follows:



encoder dencoder

$$h_i = f_\theta(x_i)$$

$$\hat{\mathbf{x}} = g_\theta(\mathbf{h})$$

Representation learning

- The auto-encoder training consists in finding the parameter set θ that minimizes the reconstruction error:

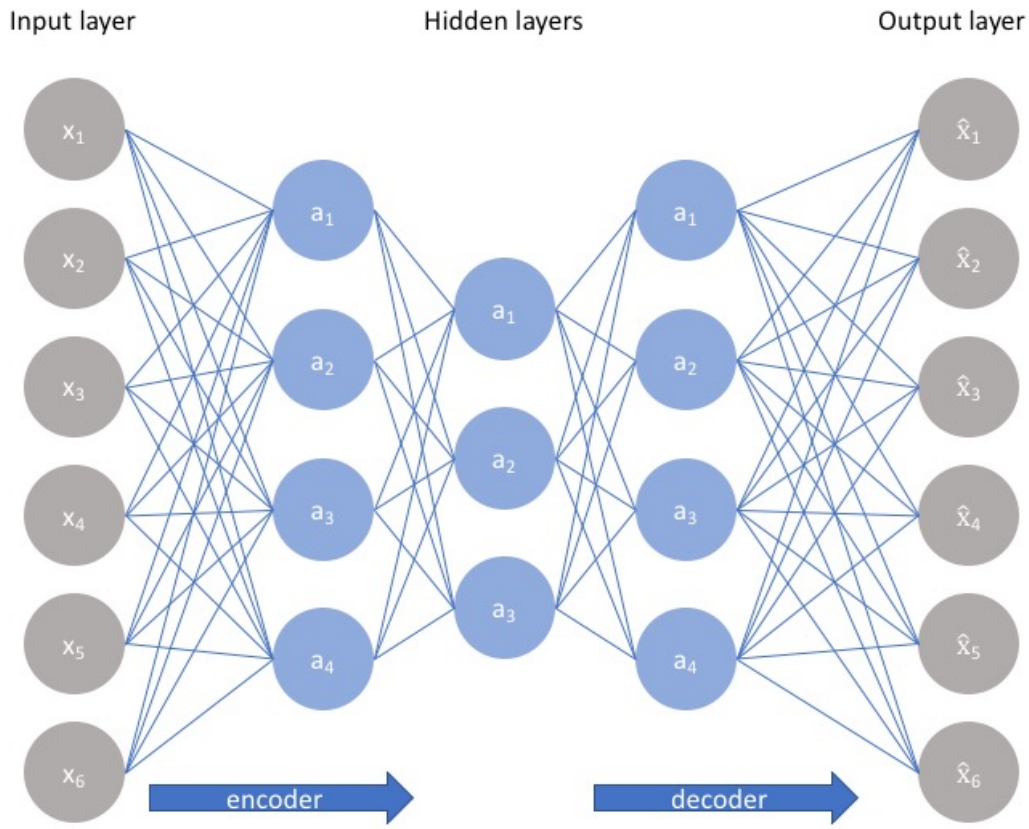
$$\mathcal{L}_{AE}(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\Big(x_i, g_\theta(f_\theta(x_i))\Big)$$

Representation learning

- **Why the «funnel» structure?**
  - If the only purpose of autoencoders was to copy the input to the output, they would be useless.
  - by training the autoencoder to copy the input to the output, the latent representation will take on useful properties.
  - This can be achieved by creating constraints on the copying task.
  - One way to obtain useful features from the autoencoder is to constraint the hidden to have smaller dimensions than x, in this case the autoencoder is called undercomplete.

Representation learning

- A typical undercomplete autoencoder has the following architecture:



REPRESENTATION LEARNING

- Sparse Autoencoders
  - They offer an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes on hidden layers.
  - It is simply an autoencoder whose training criterion involves a sparsity penalty **Ω(h)** on the hidden layer h, in addition to the reconstruction error:

$$L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(\boldsymbol{h})$$

  - where g is the decoder output

Representation learning

- Sparse Autoencoders
  - it constructs a loss function in order to penalize activations within a layer.
  - For any given observation, we encourage our network to learn an encoding and decoding which only relies on activating a small number of neurons.

- One way to penalize activation within a layer is to impose a sparsity constraint:
  - measure the hidden layer activations for each training batch and adding some term to the loss function in order to penalize excessive activations.

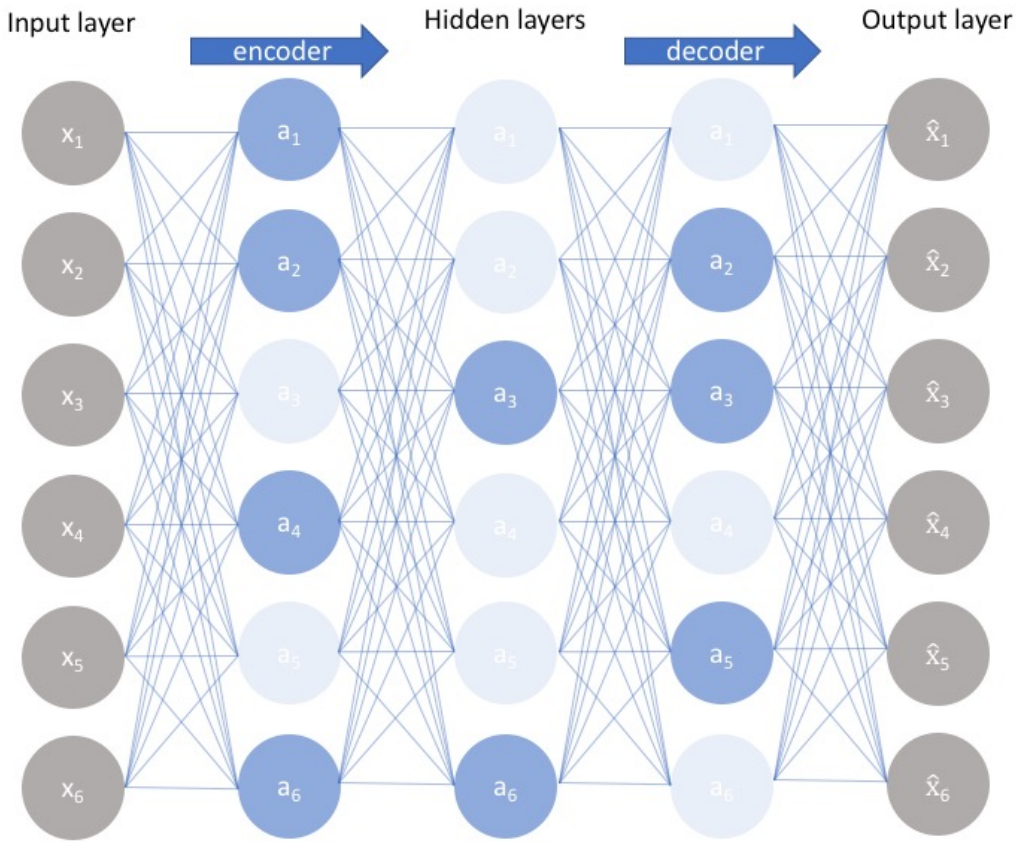
Representation learning

- L1 Regularization: We can add a term to our loss function that penalizes the absolute value of the vector of activations a in layer h for observation i, scaled by a tuning parameter.

$$\mathcal{L}(x, \hat{x}) + \lambda \sum_i \left| a_i^{(h)} \right|$$

Representation learning

- Sparse Autoencoders



REPRESENTATION LEARNING

- DENOISING AUTOENCODERS
- Rather than adding a penalty Ω to the cost function that learns something useful by changing the reconstruction error term of the cost

- Traditionally, autoencoders minimize some function

$$L(\boldsymbol{x}, g(f(\boldsymbol{x})))$$

where L is a loss function penalizing g(f(x)) from being dissimilar from x

- A denoising autoencoder instead minimizes

$$L(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}})))$$

  - where x~ is a copy of x that has been corrupted by some noise.

Representation learning

The loss function of Denoising Autoencoder:

$$\min_{\mathbf{W}_1,\mathbf{W}_2,\mathbf{b}_1,\mathbf{b}_2} \sum_{\ell} \|\mathbf{x}^{(\ell)} - \hat{\mathbf{x}}^{(\ell)}\|_2^2 + \lambda\left(\|\mathbf{W}_1\|_F^2 + \|\mathbf{W}_2\|_F^2\right)$$

where

$$\mathbf{h}^{(\ell)} = \sigma(\mathbf{W}_1\tilde{\mathbf{x}}^{(\ell)} + \mathbf{b}_1)$$
$$\hat{\mathbf{x}}^{(\ell)} = \sigma(\mathbf{W}_2\mathbf{h}^{(\ell)} + \mathbf{b}_2)$$

Like deep Autoencoder, we can stack multiple denoising autoencoders layer-wisely to form a Stacked Denoising Autoencoder.

REPRESENTATION LEARNING

- Denoising Autoencoder