Introduction to Deep Learning

Lecture 3
Deep Learning Optimization

Maria Vakalopoulou & Stergios Christodoulidis

MICS Laboratory

CentraleSupélec
Université Paris-Saclay

Friday, 24 November, 2023

Last Lecture

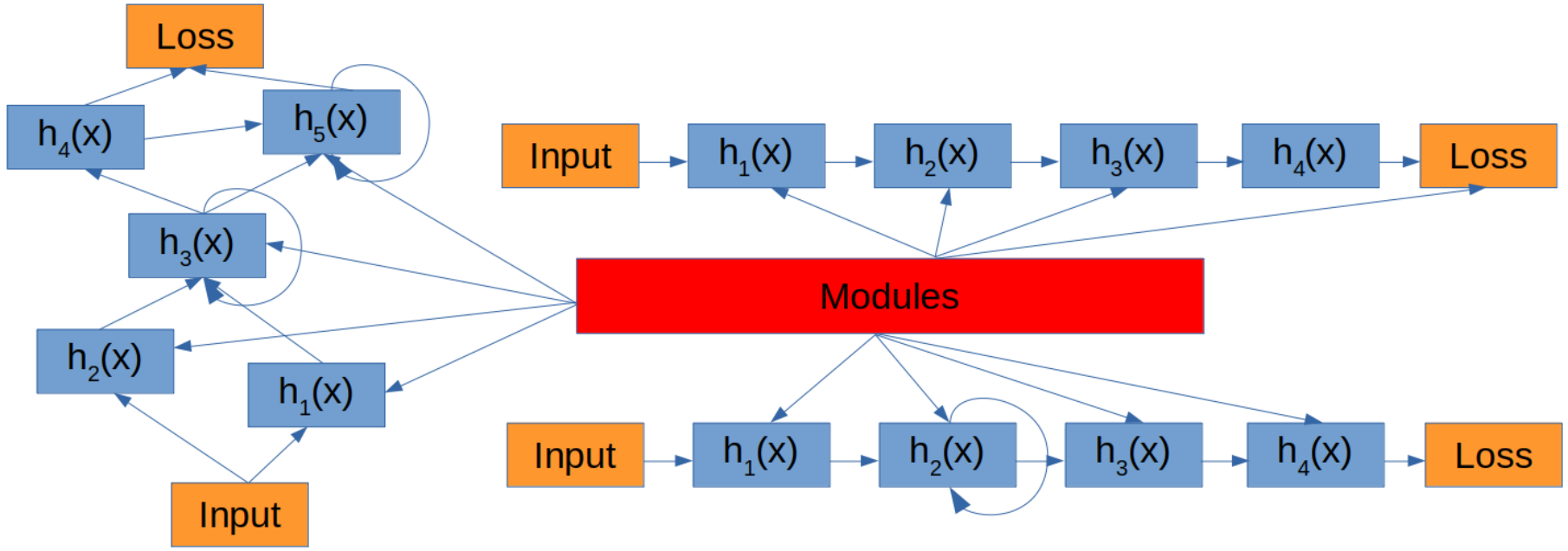
Deep Learning – The main idea

- A family of parametric, non linear and hierarchical representation learning functions, which are massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.
- $a_L(x;\theta_{1,...L}) = h_L(h_{L-1}(...h_1(x,\theta_1),\theta_{L-1}),\theta_L)$
  - X: input, θ_L: parameters for layer l, $a_L = h_L(x, \theta_L)$: (non)linear function
- Given train in corpus {X,Y} find optimal parameters

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,...,L}))$$

Many, many modules .....

- Linear, non linear functions (ReLU, sigmoid, tanh, softmax...)
- Loss functions (cross entropy, mean square error, ...)



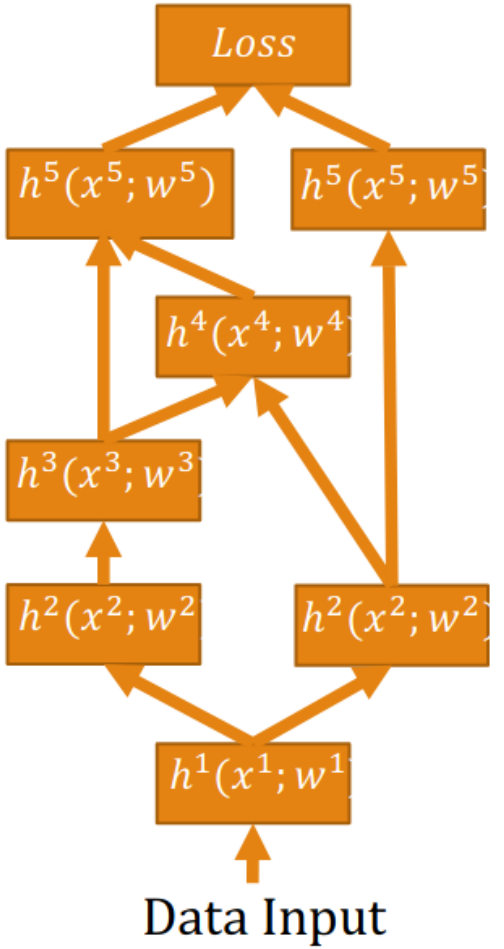
New modules

- Everything can be a module, given some rules
- How to make our own module?
  - Write a function that follows the rules
- Needs to be (at least) first-order differentiable (almost) everywhere
- Hence, we need to be able to compute the

$$\frac{\partial a(x;w)}{\partial x} \text{ and } \frac{\partial a(x;w)}{\partial w}$$

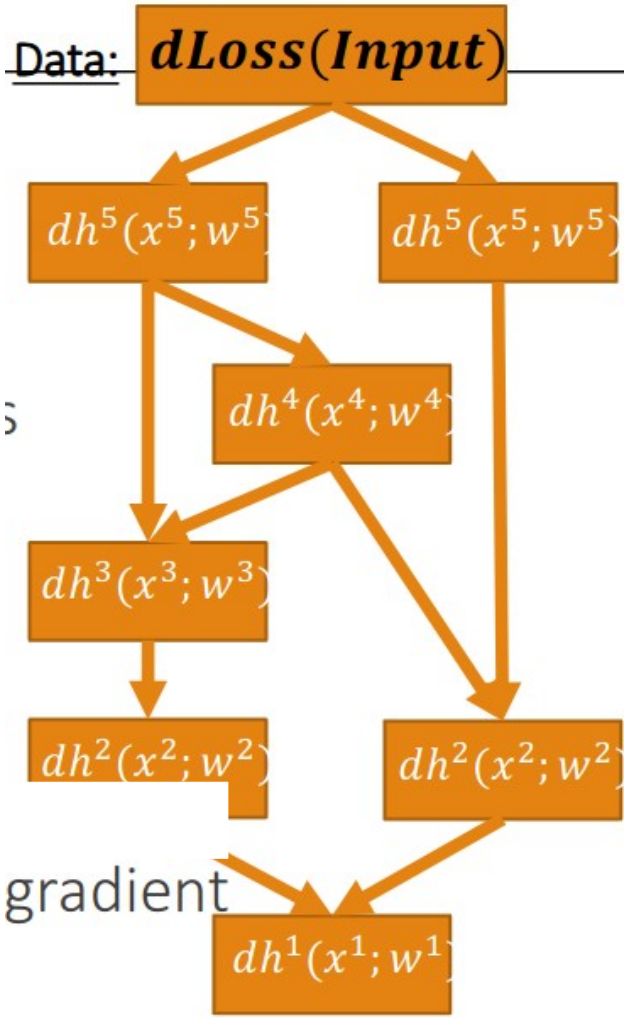
Forward graph

- Simply compute the activation of each module in the network $a^l = h^l(x^l; w)$
  - Then, set $x^{l+1} := a^l$
  - Must know the precise function behind each module $h^l(\dots)$
- Then repeat recursively
  - Visit modules one by one starting from the data input
  - One module's output is another module's input
  - Some modules might have several inputs from multiple modules
  - Store intermediate values → save compute time at the cost of memory
- Compute modules activations with the right order
  - Make sure all the inputs are computed at the right time



Backward graph

- Same story but in reverse
  - Take the reverse network (reverse connections) and go backwards
  - Instead of activation functions, use gradients of activation functions
- Requirements
  - Must know the gradient formulation of each module $\partial h^l(x^l; w^l)$
  - w.r.t. both their inputs x^l and parameters w^l
  - Note: We need the forward computations first. Activations are part of the gradients, including the final loss and its gradient. The total loss is the sum of losses for all inputs in our batch
- The whole process can be described very neatly and concisely with the backpropagation algorithm



Today’s Lecture

Today’s Lecture

- How to define our model and optimize it in practice
- Optimization methods
  - Gradient Descent
  - Stochastic Gradient Descent
- Data preprocessing and normalization
- Regularizations
- Learning rate
- Data Augmentation
- Weight initializations
- Good practices

A neural/ Deep Network in a nutshell

- The Neural Network

$$a_L(x; w_{1,\dots,\mathrm{L}}) = h_L\left(h_{L-1}(\dots h_1(x, \mathrm{w}_1), \mathrm{w}_{L-1}), \mathrm{w}_L\right)$$

- Learning by minimizing the error

$$\mathrm{w}^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; w_{1,\dots,\mathrm{L}}))$$

- Optimizing with Gradient Descent based methods

$$w_{t+1} = w_t - \eta_t \nabla_w \mathcal{L}$$

What is a difference between Optimization and Machine Learning?

1) The optimal machine learning solution is not necessarily the optimal solution

2) They are practically equivalent

3) Machine learning relates to optimization, with some differences

4) In learning we usually do not optimize the intended task but an easier surrogate one

5) Optimization is offline while Machine Learning can be online

What is a difference between Optimization and Machine Learning?

1) The optimal machine learning solution is not necessarily the optimal solution

2) They are practically equivalent

3) Machine learning relates to optimization, with some differences

4) In learning we usually do not optimize the intended task but an easier surrogate one

5) Optimization is offline while Machine Learning can be online

Pure Optimization vs Machine Learning Training?

- Pure optimization has a very direct goal: finding the optimum
  - Step 1: Formulate your problem mathematically as best as possible
  - Step 2: Find the optimum solution as best as possible
  - E.g., optimizing the railroad network in France
    - Goal: find optimal combination of train schedules, train availability, etc
- In Machine Learning, instead, the real goal and the trainable goal are quite often different (but related)
  - Even “optimal” parameters are not necessarily optimal ← Overfitting …
  - E.g., You want to recognize cars from bikes (0-1 problem) in unknown images, but you optimize the classification log probabilities (continuous) in known images.

Empirical Risk Minimization

- We ideally should optimize for $\min_{\mathrm{w}} \mathrm{E}_{x,\mathrm{y}\sim p_{\mathrm{data}}}[\mathcal{L}(\mathrm{w};\mathrm{x},\mathrm{y})]$
- i.e. the expected loss under the true underlying distribution

but we do not have access to this distribution

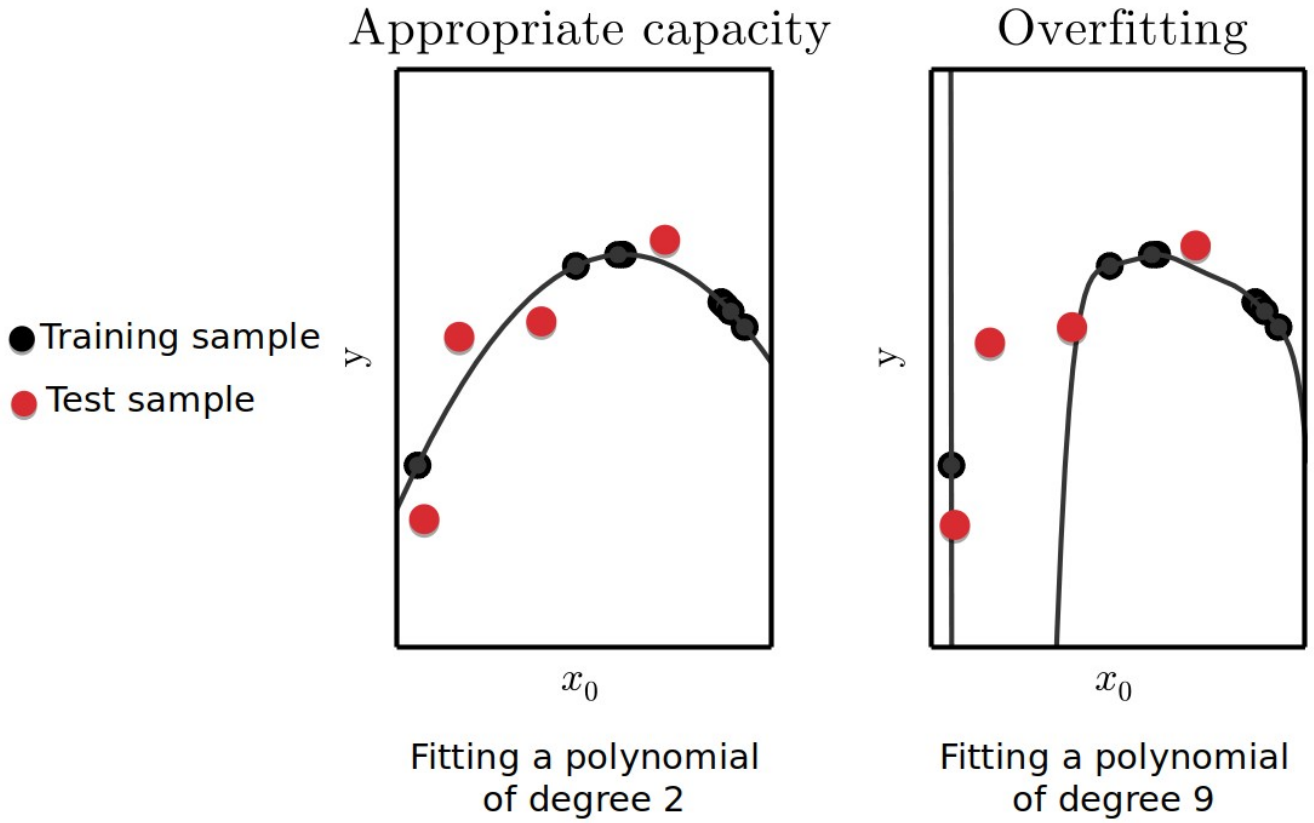
- Thus, borrowing from optimization, the best way we can get satisfactory solutions is by minimizing the empirical risk

$$\min_{w} \mathrm{E}_{x,\mathrm{y}\sim \hat{p}_{\mathrm{data}}}[\mathcal{L}(\mathrm{w};\mathrm{x},\mathrm{y})] = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(h(x_i;w),y_i)$$

- That is, minimize the risk on the available training data.

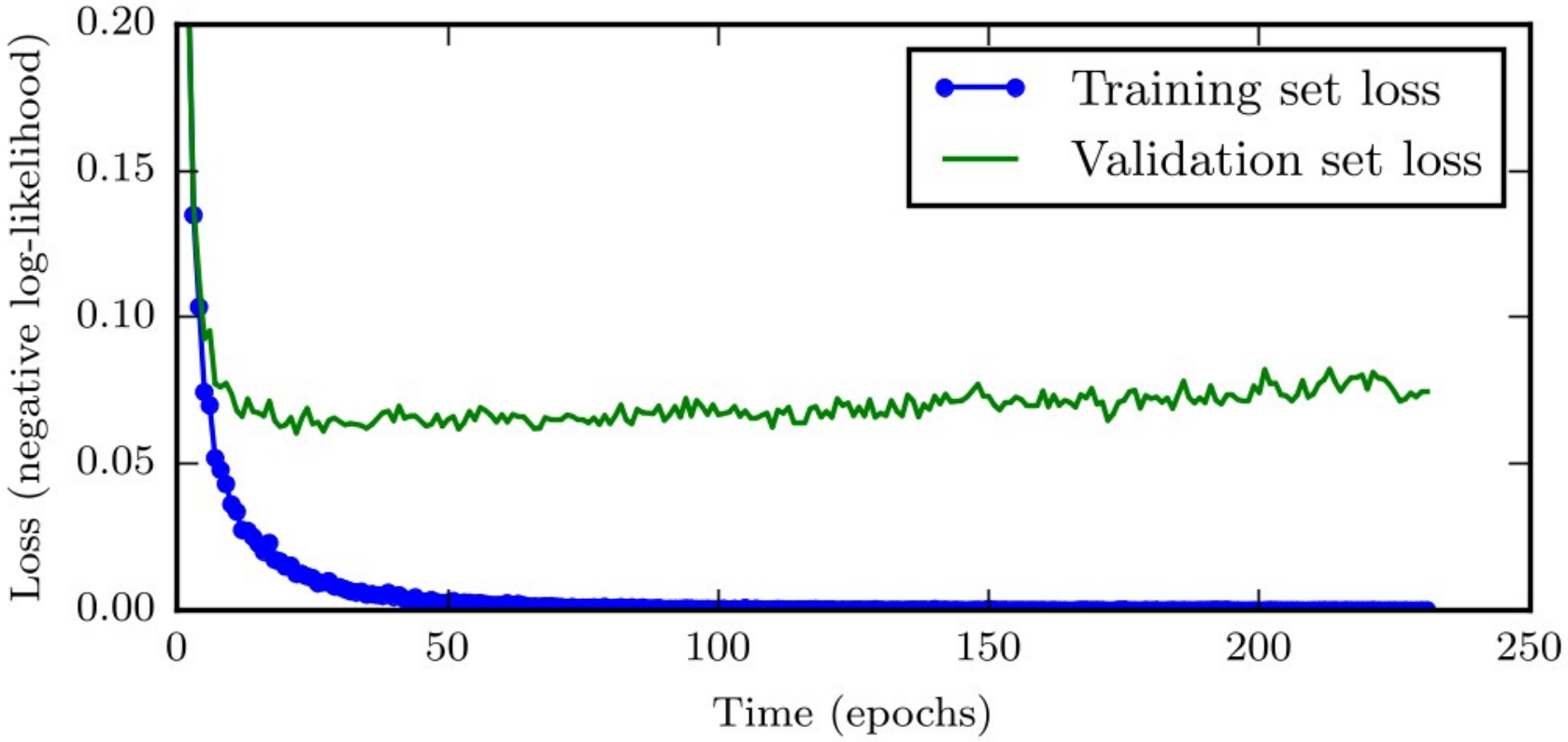
Generalization and Overfitting

- The loss function can be small on the training data, but large on the test data:
  - Overfitting



Preventing Overfitting

- Use a validation set: Monitor the loss on the validation set during training, stop when it starts increasing:



Stochastic Gradient Descent

Gradient Descent

- To optimize a given loss function, most machine learning methods rely on Gradient Descent and variants

$$w_{t+1} = w_t - \eta_t g_t$$

  - Gradient $g_t = \nabla_t \mathcal{L}$
- Gradient on full training set

$$g_t = \frac{1}{m} \sum_{i=1}^{m} \nabla_w \mathcal{L}\left(w; x_i, y_i\right)$$

  - Compute empirically from all available training samples (x_i, y_i)
  - Sample gradient → Only an approximation to the true gradient g_t^* if we knew the real data distribution

Stochastic Gradient Descent

- Introduce a second approximation in computing the gradients → SGD
  - Stochastically sample “mini-training” sets (“mini-batches”) from dataset D

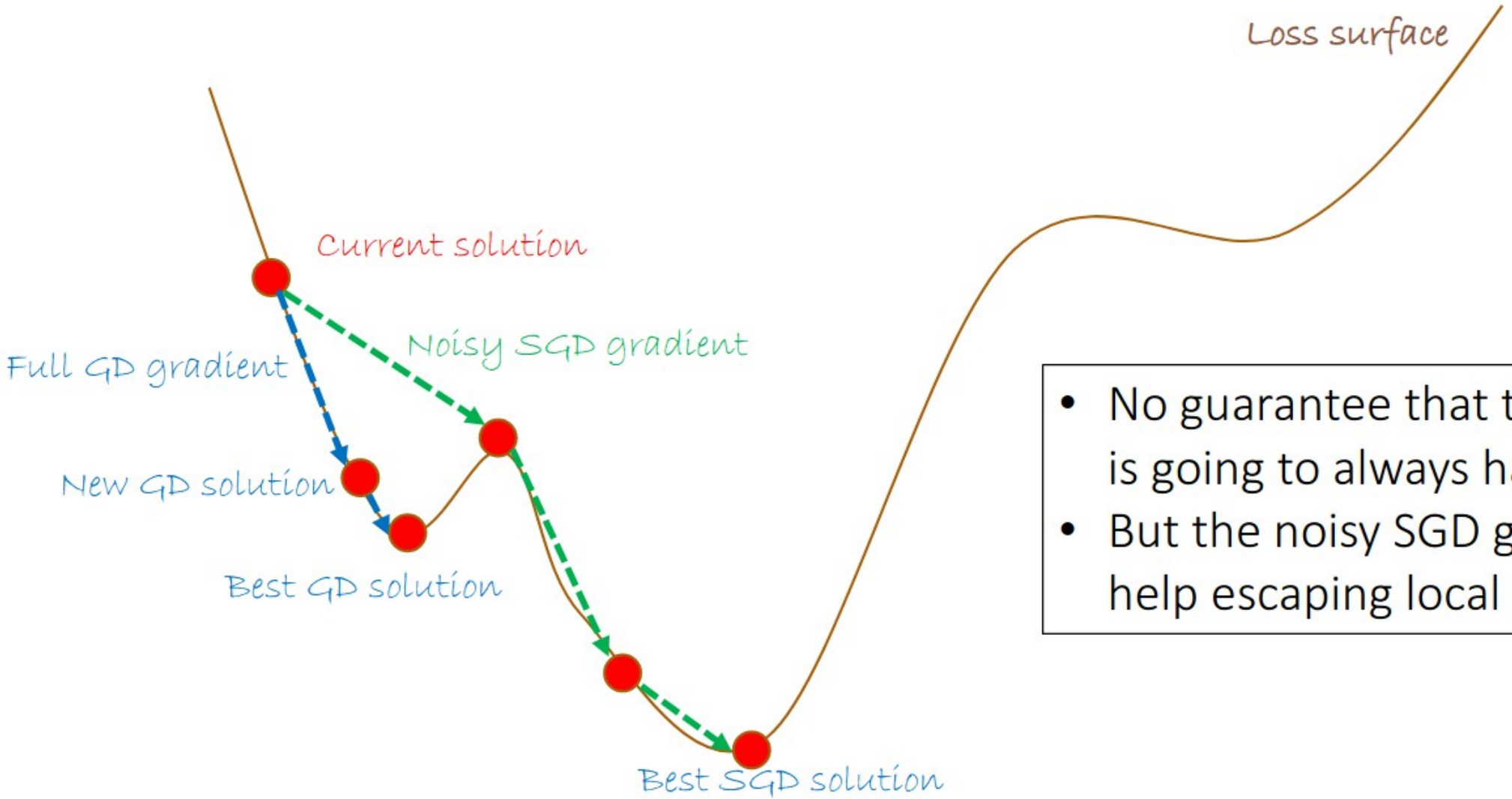
$$B_j = sample(D)$$

$$w_{t+1} = w_t - \frac{\eta_t}{|B_j|} \sum_{i \in B_j} \nabla_w \mathcal{L}_i$$

Some advantages of SDG

- Randomness helps avoid overfitting solutions
  - Variance of gradients increases when batch size decreases
- In practice, accuracy is often better
- Much faster than Gradient Descent
- Suitable for datasets that change over time

SDG is often better



- No guarantee that this is what is going to always happen.
- But the noisy SGD gradients can help escaping local optima

SDG helps avoid overfitting

- Gradient Descent: Complete gradients fit optimally the (arbitrary) data we have, not necessarily the distribution that generates them
  - All training samples are the “absolute representative” of the input distribution
  - Suitable for traditional optimization problems: “find optimal route”
  - But for ML we cannot make this assumption → test data are always different
- SGD: sampled mini-batches produce roughly representative gradients
  - Model does not overfit (as much) to the particular training samples

SDG and convergence

- How do we know that we will converge to a local minimum?

Theorem: Suppose a function $\mathcal{L}$ is convex and differentiable and that $\mathbf{E}\|G(\Theta^t)\|^2 < C^2$ together with $\mathbf{E}\|\Theta^t - \Theta^*\| < D^2$. Then for the stochastic gradient descent method it holds $\eta_t = \frac{c}{\sqrt{t}}$ that

$$\mathcal{L}(\bar{\Theta}^t) - \mathcal{L}(\Theta^*) \leq \frac{c^{-1}D^2 + c\sqrt{\frac{t+1}{t}}C^2}{\sqrt{t}}$$

where $\bar{\Theta}^t = \frac{1}{t}\sum_{i=0}^{t}\Theta_t$

Shuffling examples

- Applicable only with SGD
- Choose samples with maximum information content
  - Mini batches should contain examples from different classes
  - Prefer samples likely to generate larger errors
    - Otherwise gradients will be small → slower learning
    - Check the errors from previous rounds and prefer “hand examples”
    - Beware of outliers
- In practice, split your dataset into mini-batches
  - New epoch → create new randomly shuffled batches

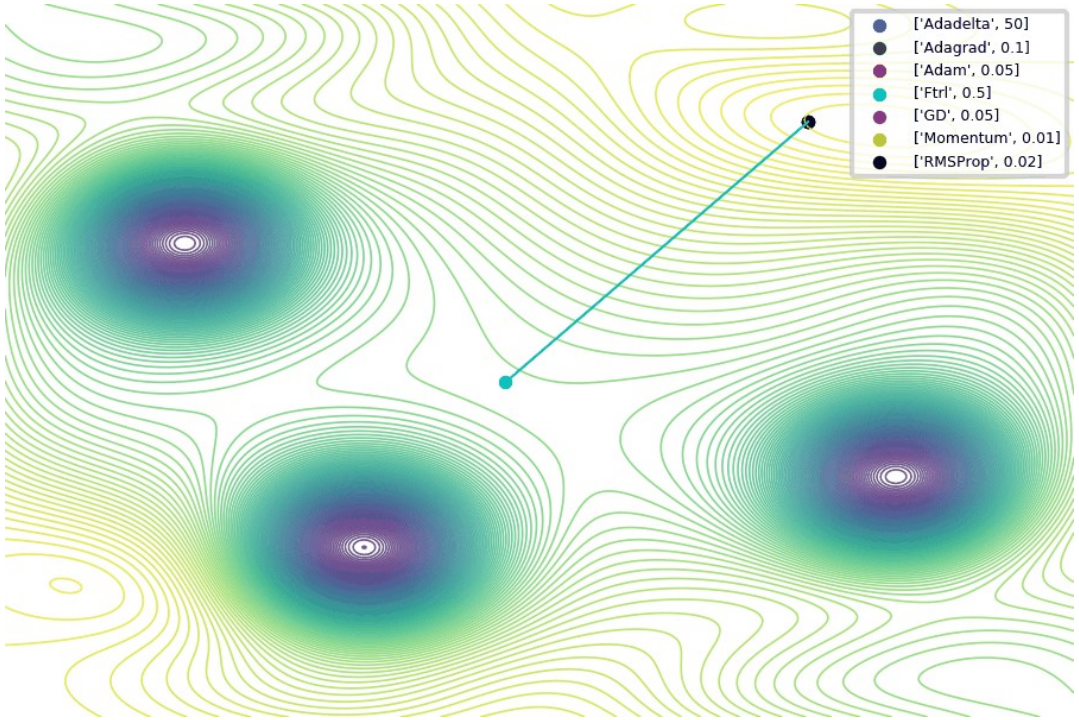
In practice

- SGD is preferred to Gradient Descent
- Training is orders of magnitude faster
  - In real datasets Gradient Descent is not even realistic
- Solutions generalize better
  - Noisier gradients can help escape local minima
  - More efficient → larger datasets → better generalization
- How many samples per mini-batch?
  - Hyper-parameter, trial & error
  - Usually between 32-256 samples
  - A good rule of thumb → as many as your GPU fits

	GD	SGD
Iteration cost	$O(n)$	$O(1)$
Convergence rate	$O(1/k)$	$O(1/\sqrt{k})$

Advanced Optimizations

Using different optimizers



Jaewan-Yun

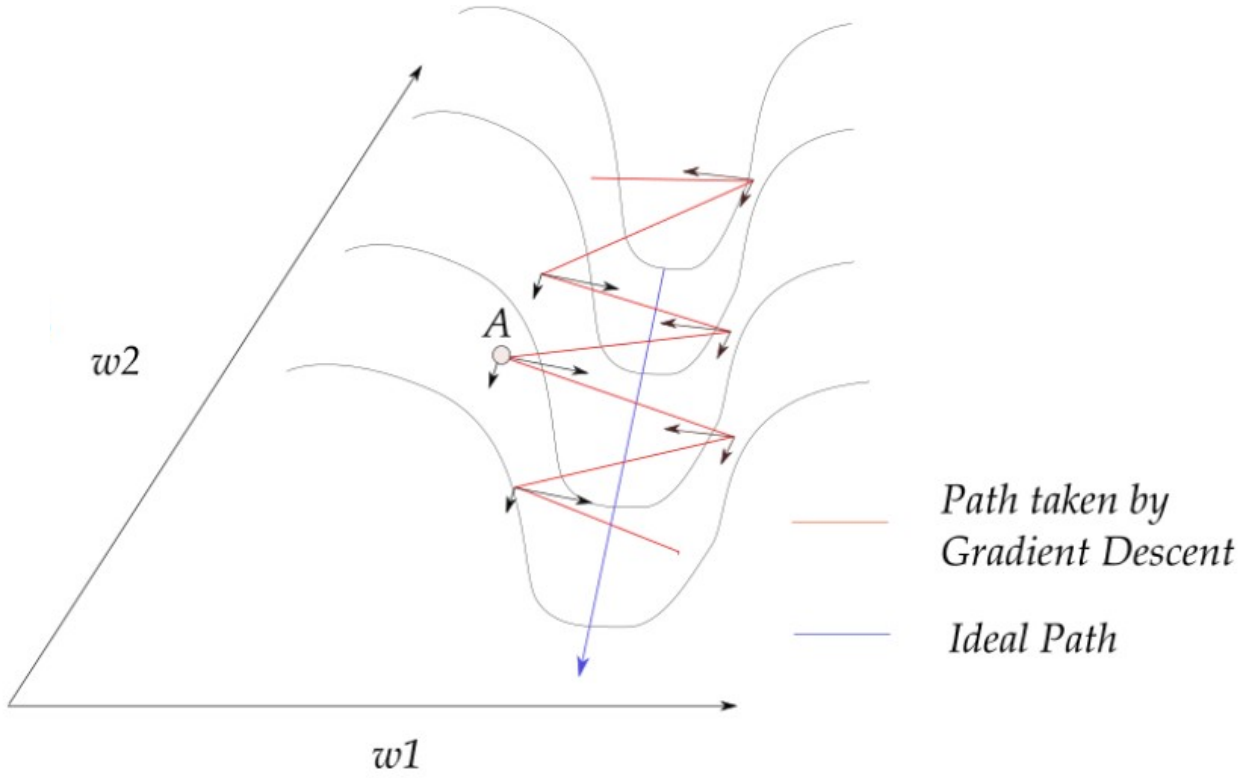
Momentum

- Do not switch update direction all the time
- Maintain “momentum” from previous update

$$u_{t+1} = \gamma u_t - \eta_t g_t$$

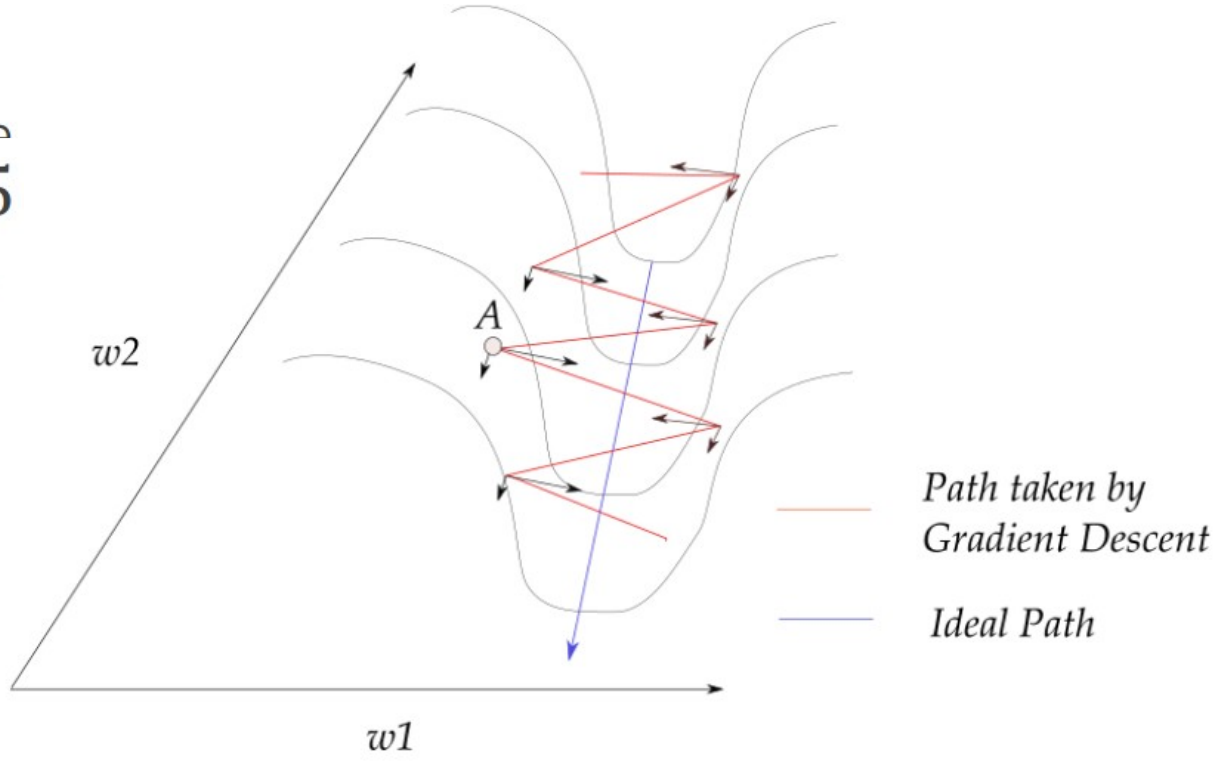
$$w_{t+1} = w_t + u_{t+1}$$

- Expon
  - With $\gamma = 0.9$ and $u_0 = 0$
  - $u_1 \propto -g_1$
  - $u_2 \propto -0.9g_1 - g_2$
  - $u_3 \propto -0.81g_1 - 0.9g_2 - g_3$



Momentum

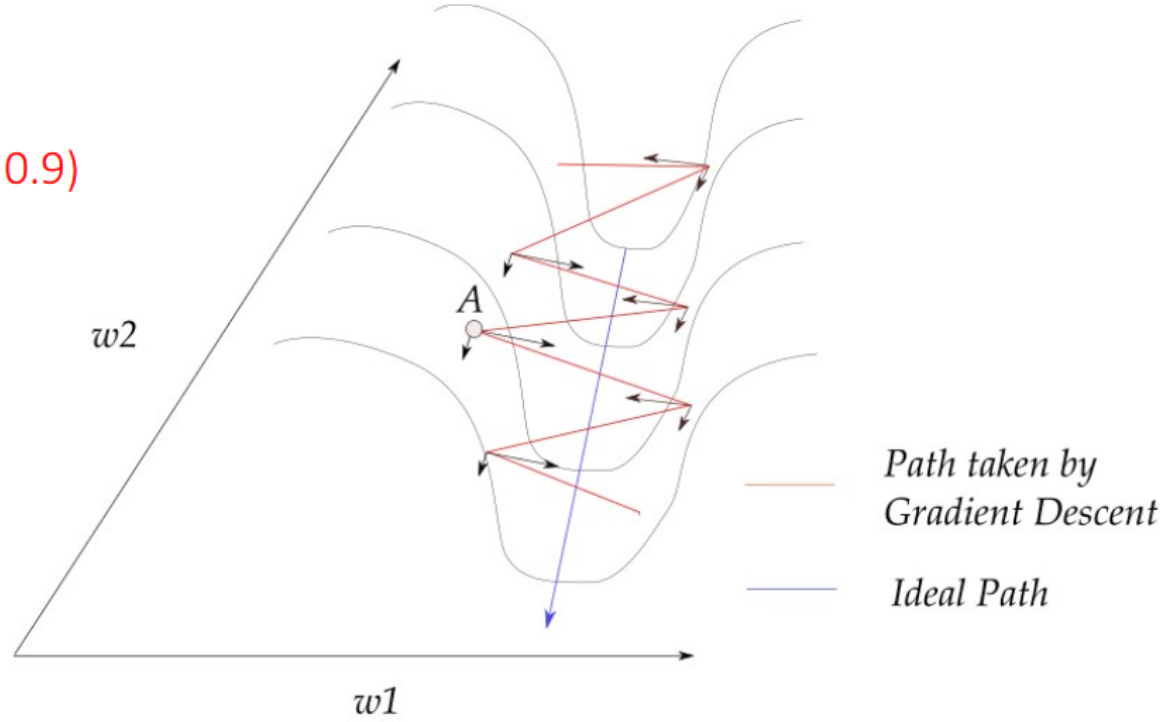
- The exponential averaging
  - Cancels out the oscullating gradients
  - Gives more weight to recent updates
- More robust gradients and learnir
  → faster convergence
- In practice, initialize $\gamma = \gamma_0 = 0.5$
  and anneal to $\gamma_\infty = 0.9$



RMSprop

Decay hyper-parameter (Usually 0.9)

- Schedule
- $r_t = \alpha r_{t-1} + (1 - \alpha) g_t^2$
- $u_t = -\frac{\eta}{\sqrt{r_t + \varepsilon}} g_t$
- $w_{t+1} = w_t + u_t$
- Large gradients, e.g. too “noisy” loss surface
  - Updates are tamed
- Small gradients, e.g. stuck in plateau of loss surface
  - Updates become more aggressive
- Sort of performs simulated annealing



Adam [Kingma2014]

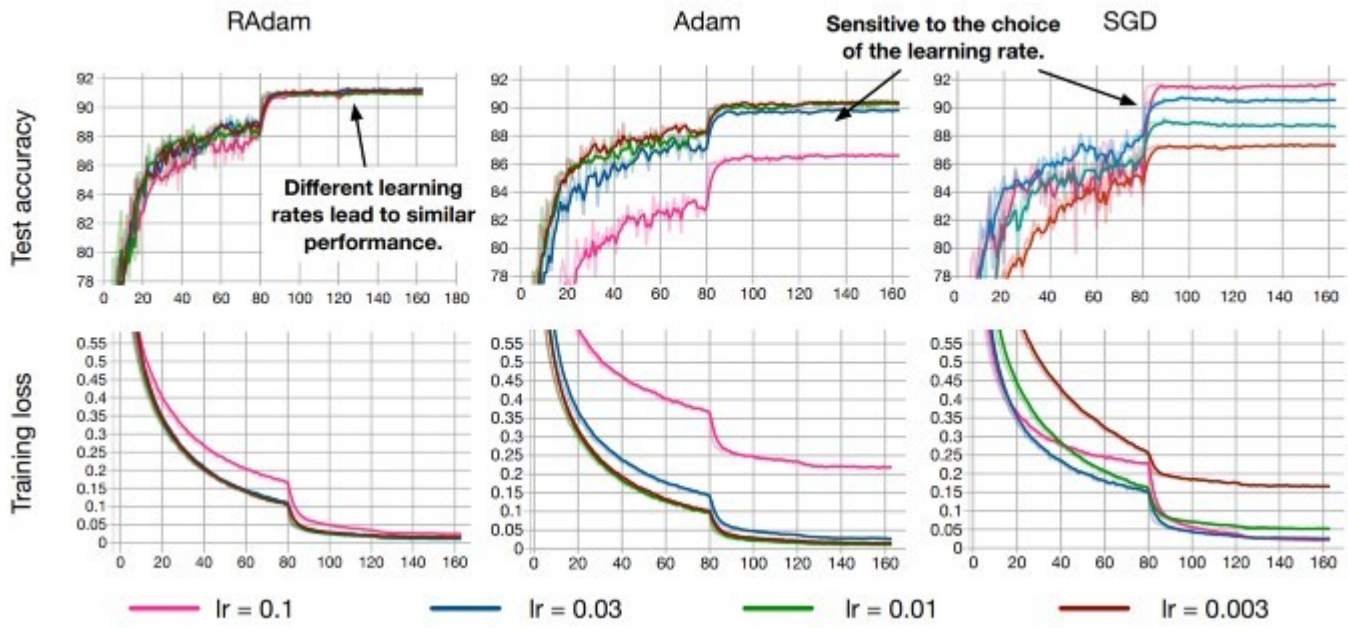
- One of the most popular learning algorithms

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$u_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t$$
$$w_{t+1} = w_t + u_t$$

  - Recommended values: $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$
- Similar to RMSprop, but with momentum & correction bias

Much much more optimizers

- Lookahead [Zhang2019]
- RAdam [Liu2019]
- Ranger [Yong2020]

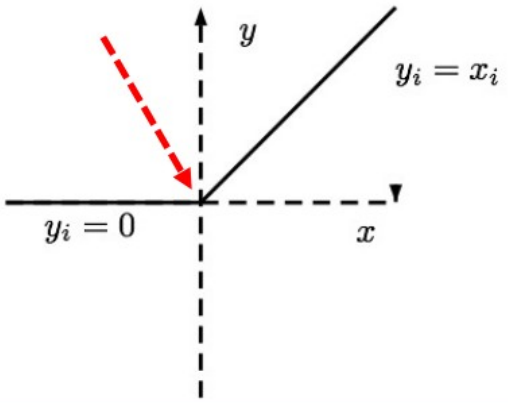


Input Normalization

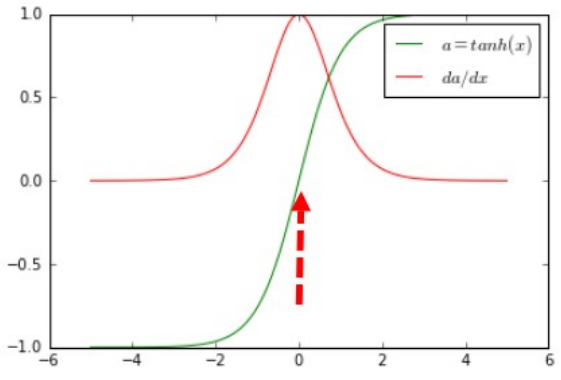
Data pre-processing

- Most common: center data roughly around 0
  - Activation functions usually “centered” around 0
    - Important for propagation to next layer: x=0 → y=0 does not introduce bias within layers (for ReLU and tanh)
    - Important for training: strongest gradients around x=0 (for tanh and sigmoid)

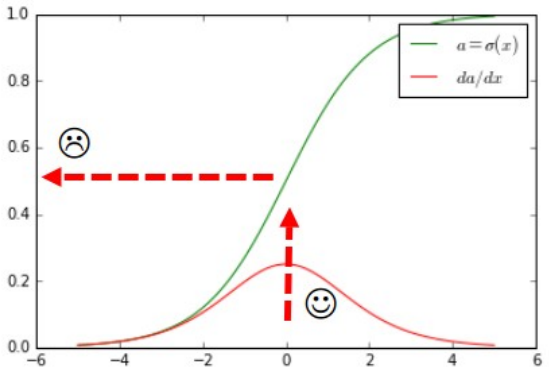
ReLU ☺



$\tanh(x)$ ☺

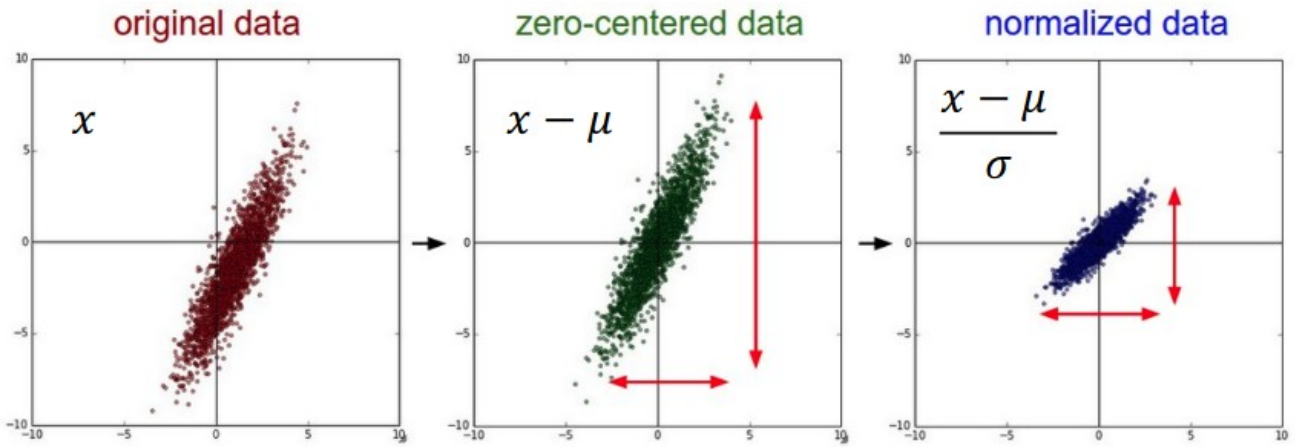


$\sigma(x)$ ☹



Unit Normalization: $N(\mu,\sigma^2) \rightarrow N(0,1)$

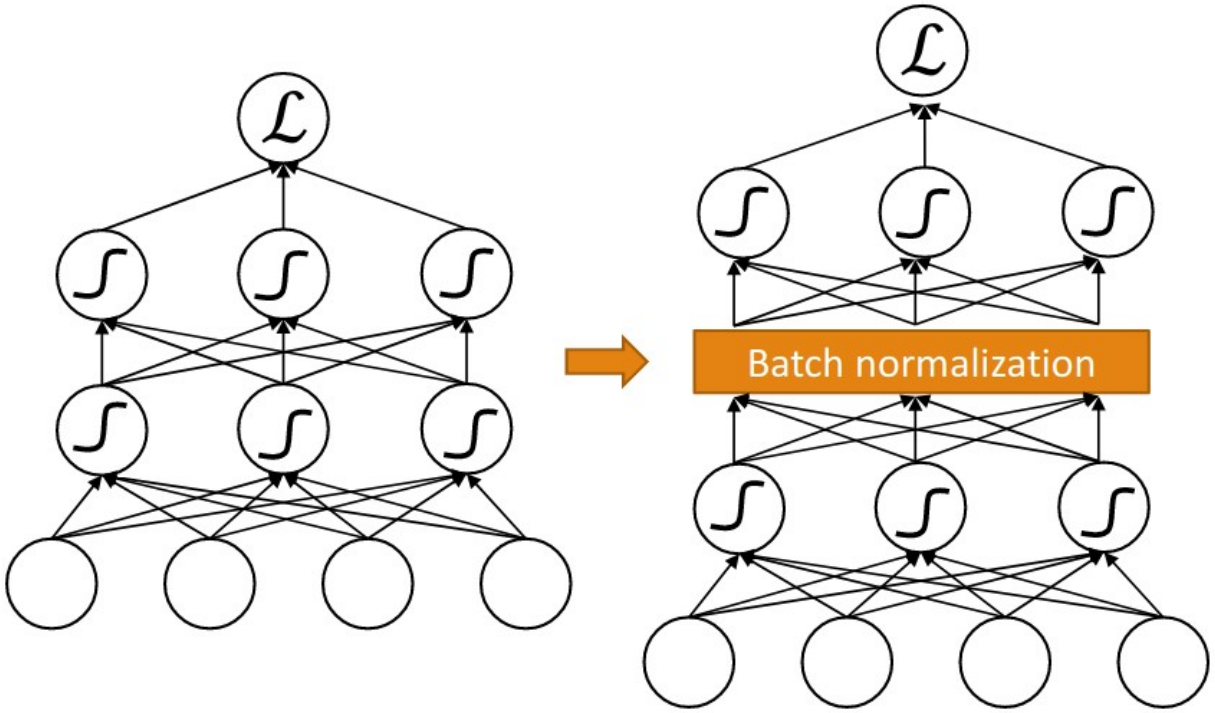
- Assume: Input variables follow a Gaussian distribution (roughly)
- Normalize by:
  - Computing mean and standard deviation from training set
  - Subtracting the mean from training/validation/testing samples and dividing the result by the standard deviation



Picture credit: Stanford Course

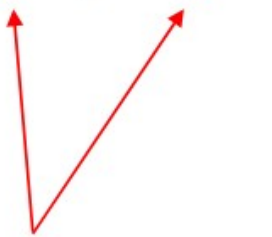
Batch normalization [Ioffe2015]

- Input distributions change for every layer, especially during training
- Normalize the layer inputs with batch normalization
  - Roughly speaking, normalize x_l to N(0,1), then rescale using trainable parameters



Batch normalization – The algorithm

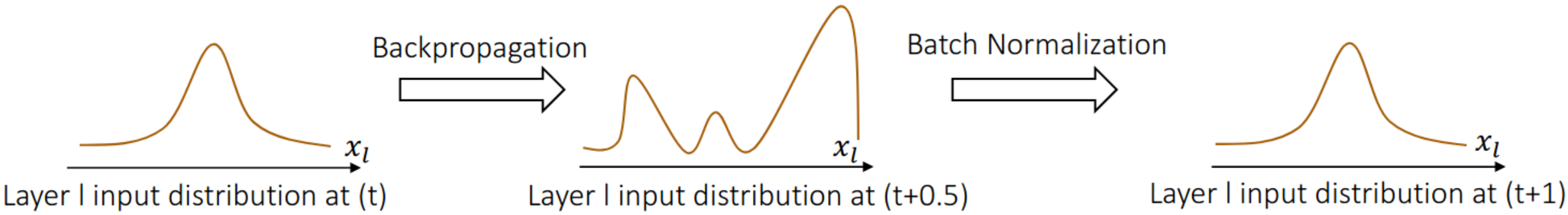
- $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ [compute mini-batch mean]
- $\sigma_B \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ [compute mini-batch variance]
- $\widehat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$ [normalize input]
- $\widehat{y_i} \leftarrow \gamma \widehat{x_i} + \beta$ [scale and shift input]



Trainable parameters

Batch normalization – Intuition

- Internal Covariate shift
  - At each step, a layer must not only adapt the weights to fit better the data
  - It must also adapt to the change of its input distribution, as its input is itself the result of another layer that changes over steps
- The distribution fed to the layers of a network should be somewhat:
  - Zero-centered
  - Constant through time and data



From training to test time

- How do we ship the Batch Norm layer after training?
  - We might not have batches at test time
- Usually: keep a moving average of the mean and variance during training
  - Plug them in at test time
  - To the limit, the moving average of mini-batch statistics approaches the batch statistics

- $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
- $\sigma_B \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
- $\widehat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$
- $\widehat{y_i} \leftarrow \gamma \widehat{x_i} + \beta$

Regularization

Regularization

- Neural Networks typically have thousands, if not millions of parameters
  - Usually, the dataset size smaller than the number of parameters
- Overfitting is a grave danger
- Proper weight regularization is crucial to avoid overfitting

$$\mathrm{w}^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; w_{1,\ldots,\mathrm{L}})) + \lambda\Omega(\theta)$$

- Possible regularization methods
  - l_2-regularization
  - l_1-regularization
  - Dropout
  - ...

l_2-regularization

- Most important (or most popular) regularization

$$w^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; w_{1,\ldots,L})) + \frac{\lambda}{2}\sum_l {w_l}^2$$

- The l_2-regularization is added to the gradient descent update rule

$$w_{t+1} = w_t - \eta_t(\nabla_\theta \mathcal{L} + \lambda w_l) \Longrightarrow$$
$$w_{t+1} = (1 - \lambda\eta_t)w^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$

- λ is usually about 10^{-1}, 10^{-2}

"Weight decay", because weights get smaller

l_1-regularization

- l_1-regularization is one of the most important regularization techniques

$$w^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a_L(x; w_{1,\dots,L})) + \frac{\lambda}{2}\sum_l |w_l|$$

- Also l_1-regularization is added to the gradient descent update rule

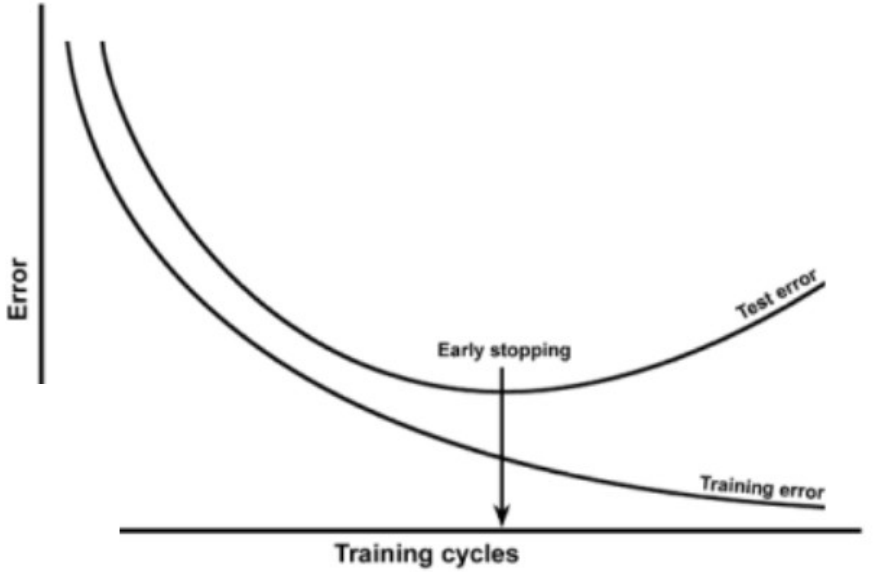
$$w_{t+1} = w_t - \eta_t \left(\nabla_\theta \mathcal{L} + \lambda \frac{w^{(t)}}{|w^{(t)}|} \right)$$

Sign function

- l_1-regularization → sparse weights
  - λ high → more weights become 0

Early stopping

- To tackle overfitting another popular technique is early stopping
- Monitor performance on separate validation set
- Training the network will decrease training error, as well validation error (although with a slower rate usually)
- Stop when validation error starts increasing
  - This quite likely means the network starts to overfit



Dropout [Srivastava2014]

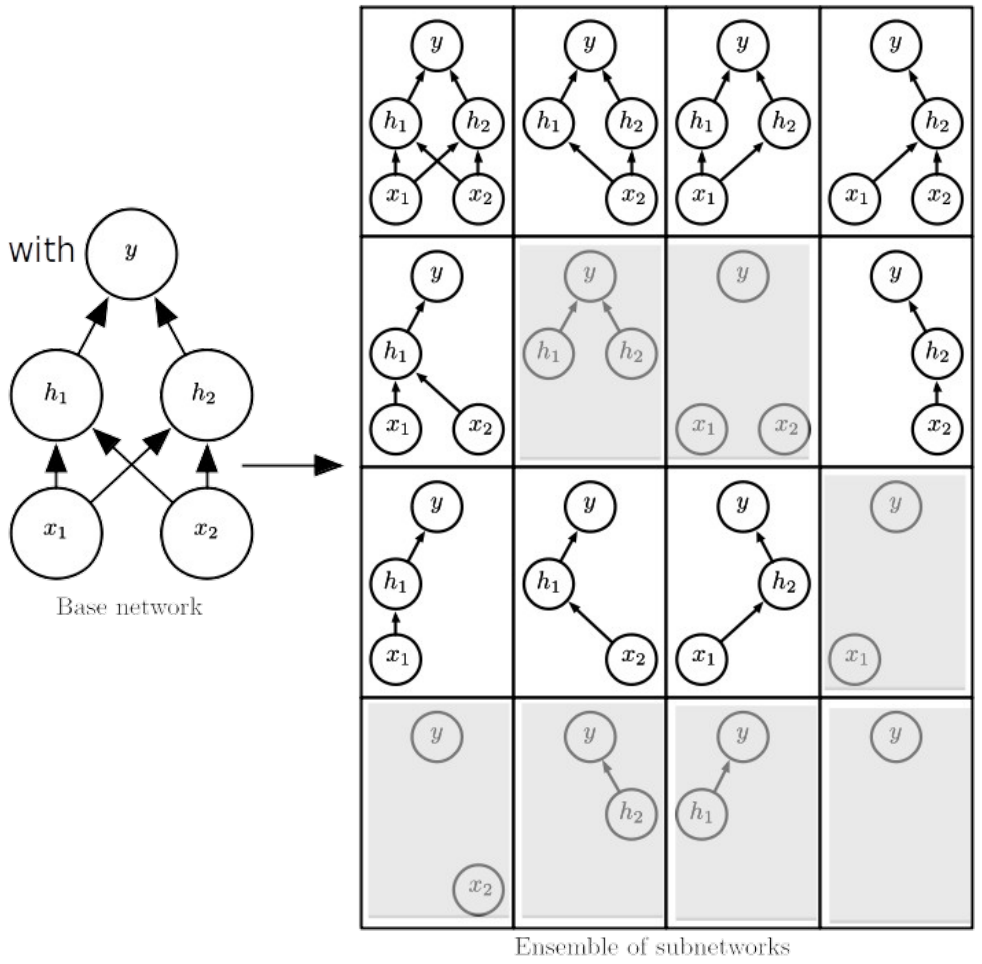
- Bagging/ Ensemble methods: Averaging different models, as different models will usually not make all the same errors on the test set (AdaBoost, Random Forests, etc.).

$$\tilde{o}(\mathbf{x}) = \frac{1}{N} \sum_{i}^{N} o_i(\mathbf{x}) \,,$$

- where the o_i are different models (classifiers, regressors, ...), and $\tilde{o}$
- the final model.
- Dropout is an ensemble method that does not need to build the models explicitly.

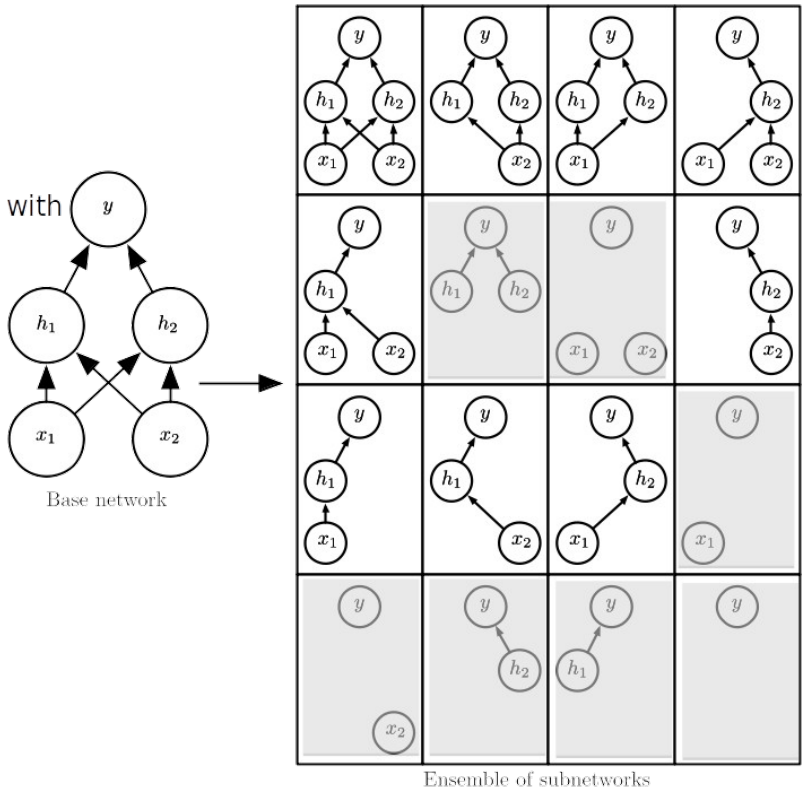
Dropout

- Considers all the networks that can be formed by removing units from a network:

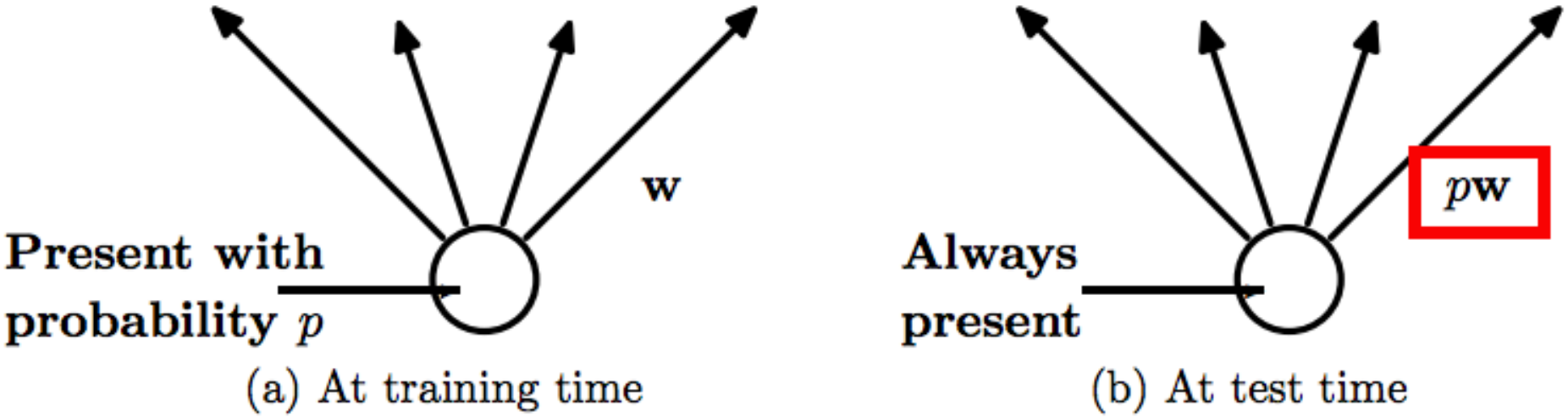


Dropout

- At each optimization iteration: random binary masks on the units to consider.
- The probability p to remove a unit is a metaparameter.



Dropout -Algorithm



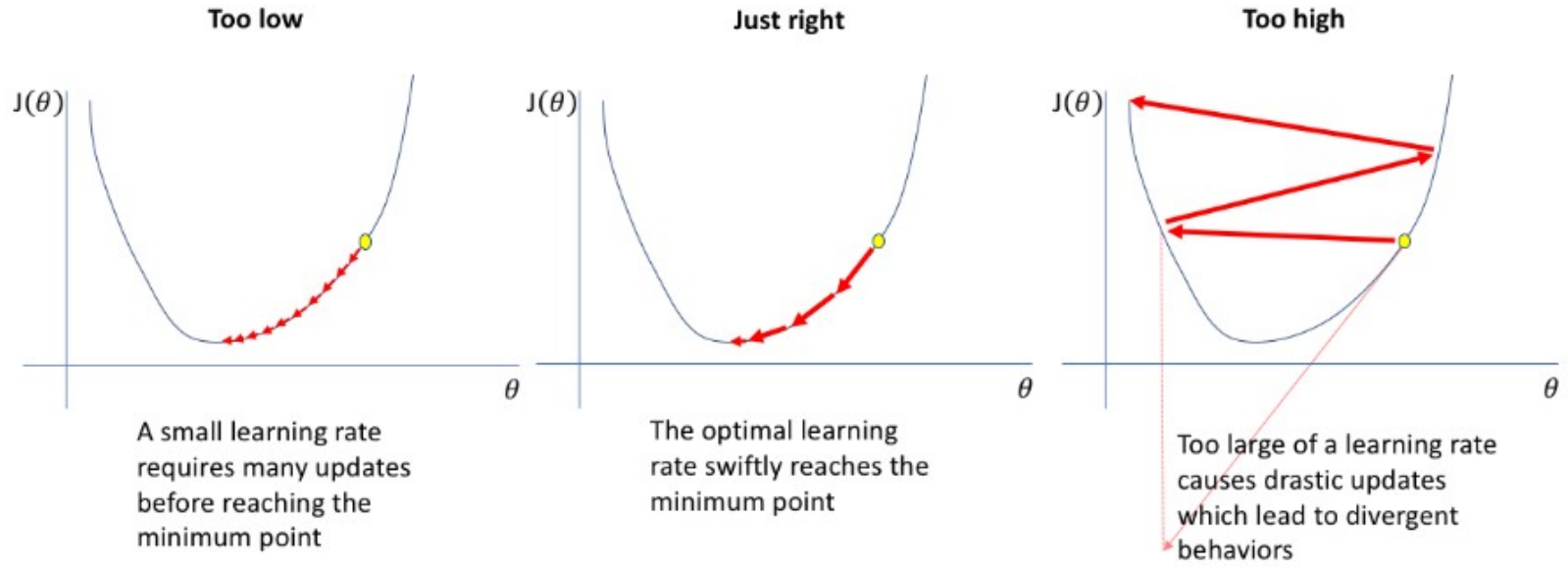
(a) At training time

(b) At test time

Learning rate

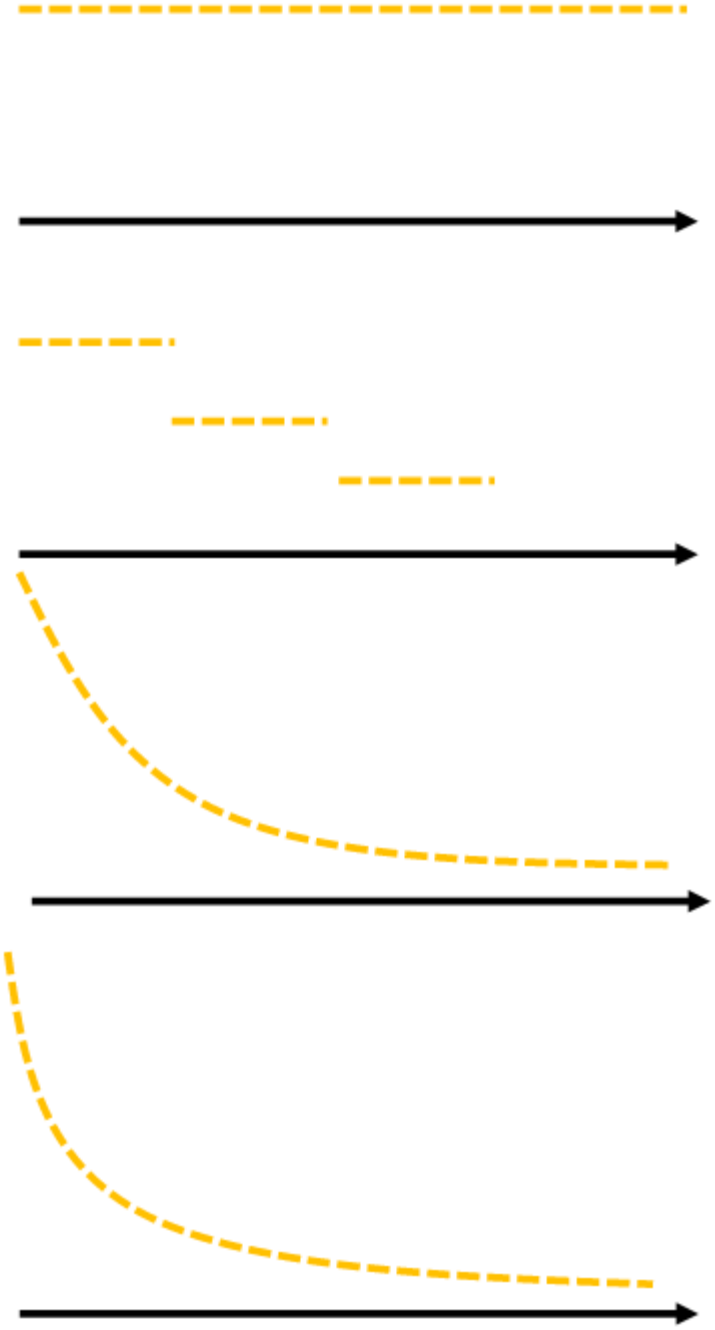
Learning rate

- The right learning rate η_t very important for fast convergence
  - Too strong → gradients overshoot and bounce
  - Too week → slow training
- Learning rate per weight Is often advantageous
  - Some weights are near convergence, others not



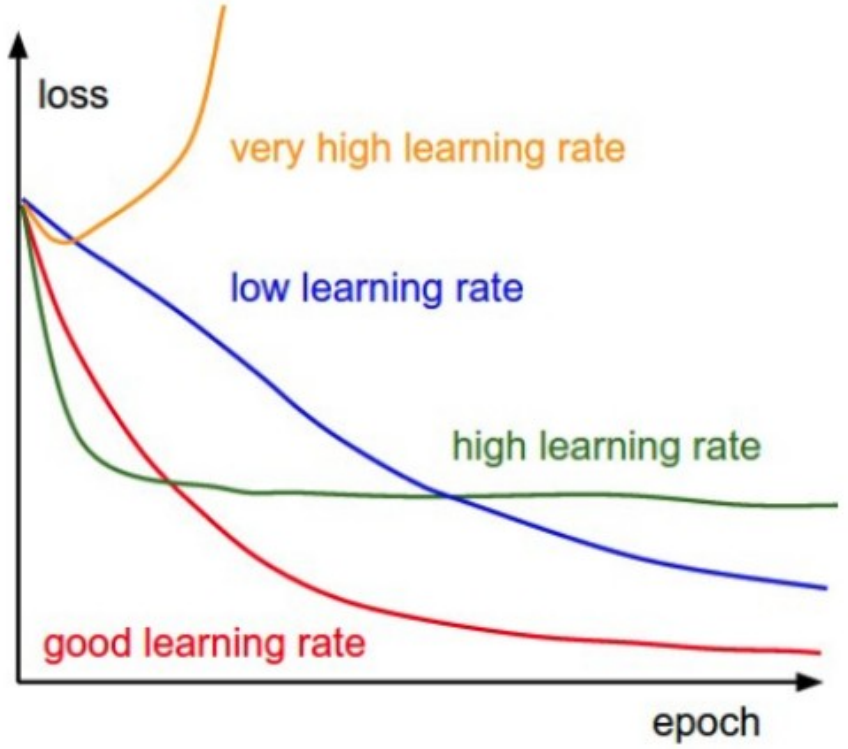
Learning rate schedules

- Constant
  - Learning rate remains the same for all epochs
- Step decay
  - Decrease every T number of epochs or when validation loss stopped decreasing
- Inverse decay $\eta_t = \frac{\eta_0}{1+\varepsilon t}$
- Exponential decay $\eta_t = \eta_0 e^{-\varepsilon t}$
- Often step decay preferred
  - Simple, intuitive, works well



In practice

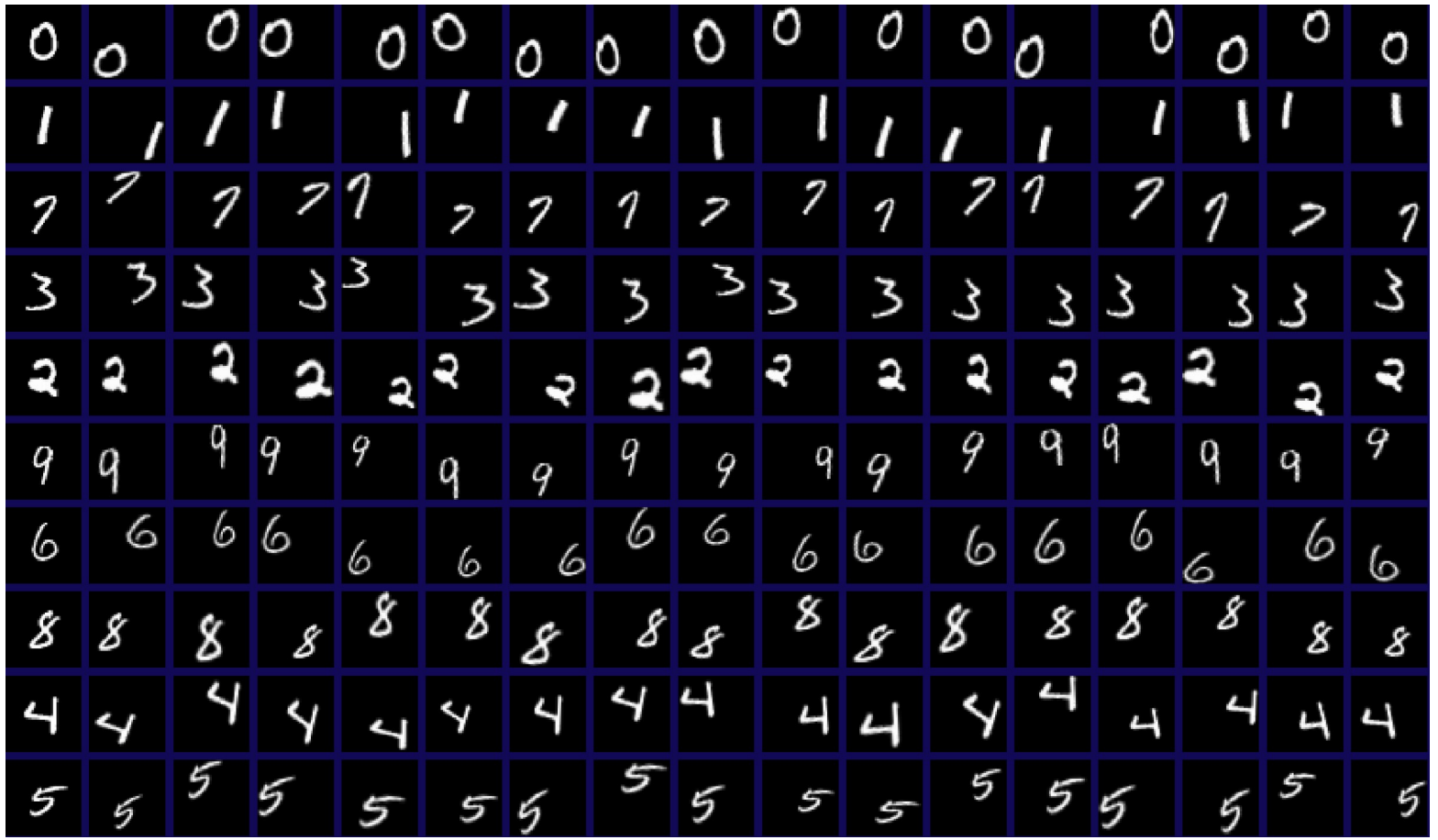
- Try several log-spaced values 10^{-1}, 10^{-2}, 10^{-3}, … on a smaller set
  - Then, you can narrow it down from there around where you get lowest validation error
- You can decrease the learning rate every 10 (or some other value) full training set epochs
  - Although this highly depends on your data



Data augmentation

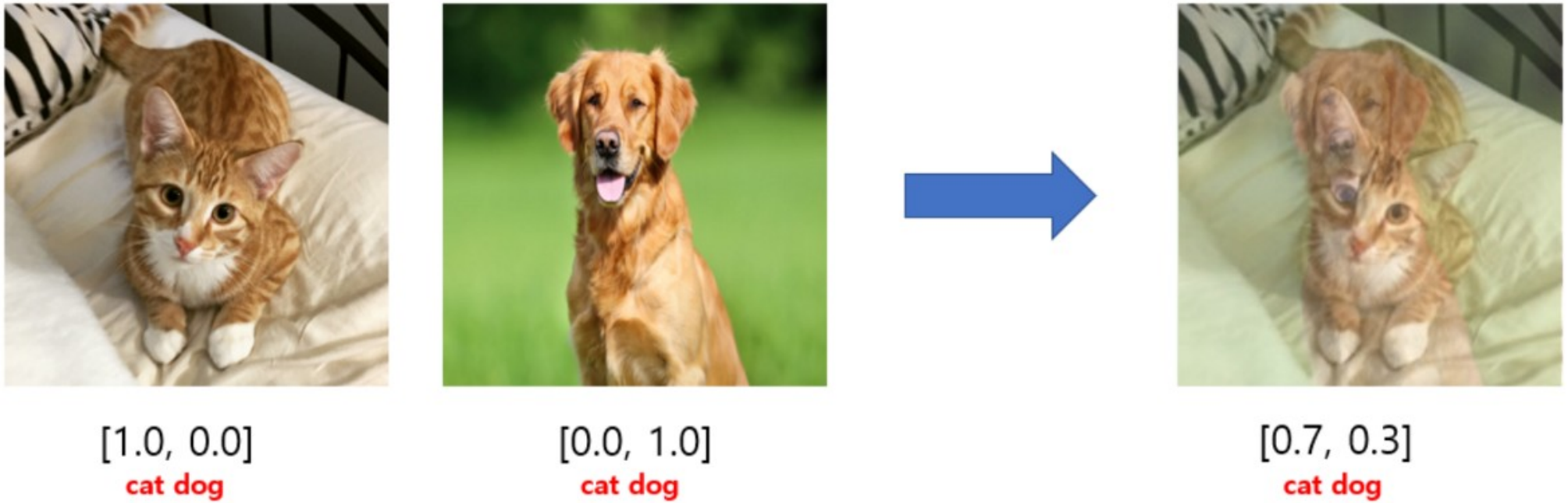
Data Augmentation

- Transform the original data, for example apply geometric transformations.



Data Augmentation

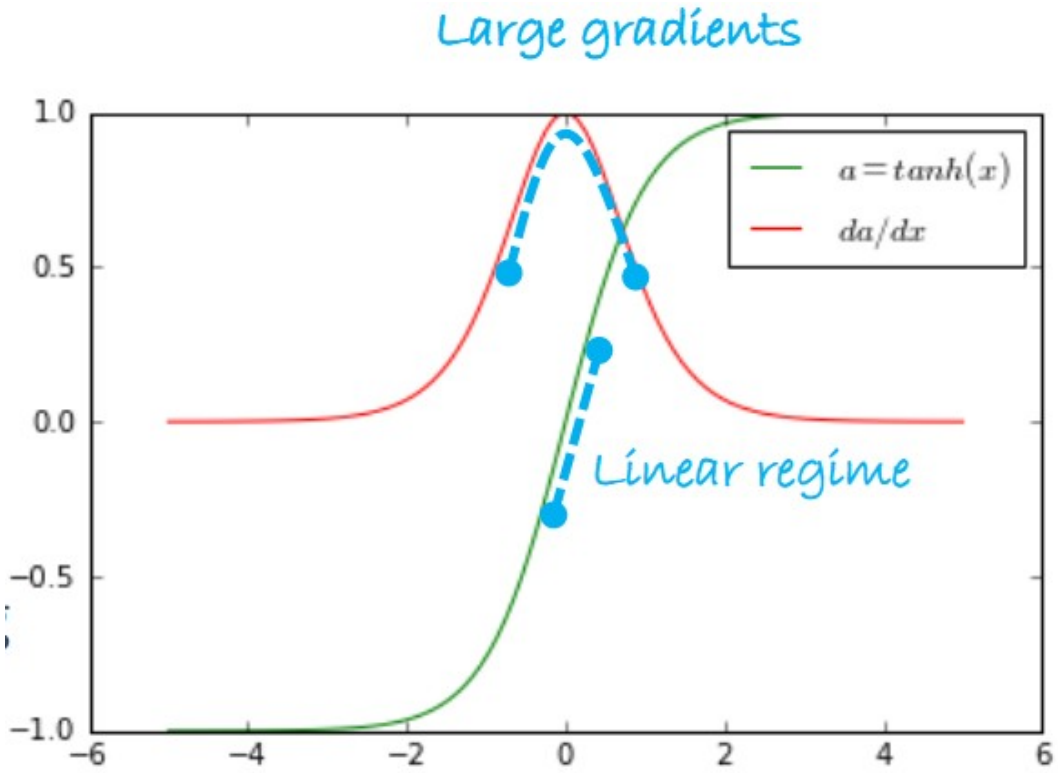
- Zhang H. et al. “mixup: Beyond Empirical Risk Minimization”. In ICLR 2018



Weight initialization

Weight initialization

- There are few contradictory requirements:
- Weights need to be small enough
  - Otherwise output values explode
- Weights need to be large enough
  - Otherwise signal is too weak for any serious learning
- Around origin (0) for symmetric functions (tanh, sigmoid)
  - When training starts, better stimulate activation functions near their linear regime
  - Larger gradients → faster training



Weight initialization

- Weights must be initialized to **preserve the variance** of the activations during the forward and backward computations

- Initialize weights to be different from one other
  - Don’t give same values to all weights (like all 0)
  - In that case all neurons generate same gradient → no learning
- Generally speaking initialization depend on
  - Non-linearities
  - Data normalization

Babysitting Deep Nets

- Always check your gradients if not computed automatically
- Check that in the first round you get loss that corresponds to random guess
- Check network with few samples
  - Turn off regularization. You should predictably overfit and get a loss of 0
  - Compare the curve between training and validation sets – there should be a gap, but not too large
- Have a separate validation set
  - Use validation set for hyper-parameter tuning
  - Compare the curve between training and validation sets – there should be a gap, but not too large
- Preprocess the data (at least to have 0 mean)
- Use regularization
- Use batch normalization