Deep Learning
SGD and regularization

Luca Bortolussi

DMG, University of Trieste, IT
Modelling and Simulation, Saarland University, DE

DSSC, Summer Semester 2018

Stochastic Gradient Descent

Gradient Descent is an iterative method to find **local minima** of $E(\mathbf{w})$ following the negative gradient direction $-\nabla E(\mathbf{w})$. Start from $\mathbf{w}_0$ and iterate:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \nabla E(\mathbf{w}_k)$$

For learning problems, $E(\mathbf{w})$ is typically the **cross-entropy** w.r.t. the empirical distribution of data:

$$E(\mathbf{w}) = 1/n \sum E_n(\mathbf{w}),$$

where the sum is over observations.

Stochastic Gradient Descent replaces $\nabla E(\mathbf{w})$ by a statistical estimate, averaging over a small number $\mathbf{m}$ of randomly chosen observations:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \sum_{n=1..m} \nabla E_n(\mathbf{w}_k)/m$$

Stochastic Gradient Descent

Stochastic Gradient Descent replaces $\nabla E(\mathbf{w})$ by a statistical estimate, averaging over a small number **m** of randomly chosen observations:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \sum_{n=1..m} \nabla E_n(\mathbf{w}_k)$$

- the m observations used to estimate $\nabla E(\mathbf{w})$ are called a **minibatch.** Typically, m=20,50,100, but m=1 can be used as well (slower convergence, online algorithm).
- if we would have a huge dataset (never use twice an observation), then SGD will give us an estimate of the gradient of the **cross-entropy w.r.t. the true data distribution**.
- Typically, datasets are not so large, and we need to pass several times through all data points: each pass is called an **epoch** of the SGD.
- At the beginning of each epoch, the dataset should be **reshuffled**, to have unbiased estimated of the gradient.

SGD: Learning Rate

The **learning rate** η_k is a crucial hyperparameter of the SGD approach. It has to be decreased during the iterations to guarantee convergence to a local minimum (due to noise in the estimation of the gradient).

Conditions for convergence: $\sum_{k=1}^{\infty} \eta_k = \infty$ and $\sum_{k=1}^{\infty} \eta_k^2 < \infty$

Typical choice: $\eta_k = (1 - \frac{k}{\tau})\eta_0 + \frac{k}{\tau}\eta_\tau$

with τ number of iterations for T epochs.
T=100 for very deep models, otherwise T=5-10.
$\eta_\tau = 0.01\eta_0$ and η_0 being set by experimenting.

The choice of η_0 is **crucial**: too large and the algorithm will diverge, too small and it will get stuck or take forever to converge.

Strategy: monitor the first 50-100 iterations, experimenting with different η_0 to find an optimal one, i.e. the one decreasing the most the error function. Choose a larger one, but not unstable.

SGD: initialisation

The initial point $\mathbf{w}_0$ of the algorithm is also a crucial parameter: we would like to start from a point in a large basin of attraction of a good minimum!
It is recommended to try multiple initial points (random restart)

Due to symmetries in the weight space, it is recommended to use an **asymmetric initial point**, to break symmetries in the weight.

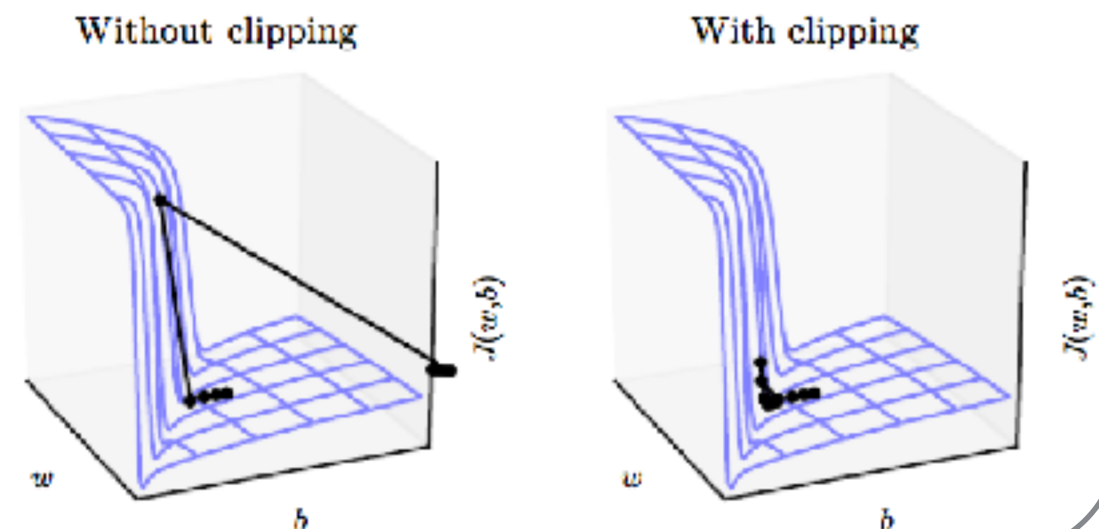
One way to achieve this is to sample $\mathbf{w}_0$ randomly. Good choices of distribution are the uniform and a Gaussian one, with zero mean. Range of the initialisation distribution is important: if it is too large, this can create instabilities (like very large gradients). If too small, variation may be too little.

Normalised initialisation for a layer with m outputs and n inputs:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

Cliffs are common for deep models

Gradient clipping: rescale gradient to a max length v.



SGD: Momentum

Gradient Descent suffers when local minima have **ill-conditioned** Hessian matrices (high curvature in near the minimum, only along some directions).

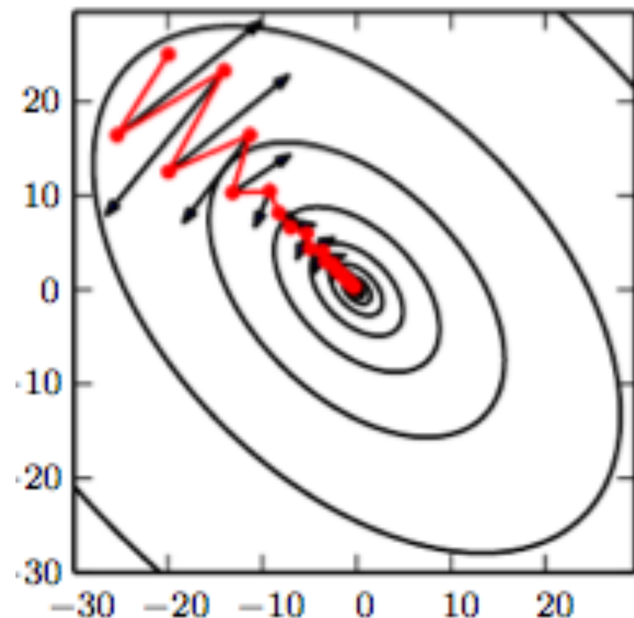
Momentum is a strategy to circumvent these problems, and helps also when **gradient estimation is noisy**.

Idea: compute an exponentially decaying average of the new gradient and the previous one. Typically, α=0.5, 0.9, 0.99

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

v can be seen as a velocity of a particle moving in the **w** space, subject to **potential** E(**w**) and to **viscous friction** (proportional to v itself).



Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

strong enough to prevent motion if the gradient does not justify moving

8.3.3 Nesterov Momentum

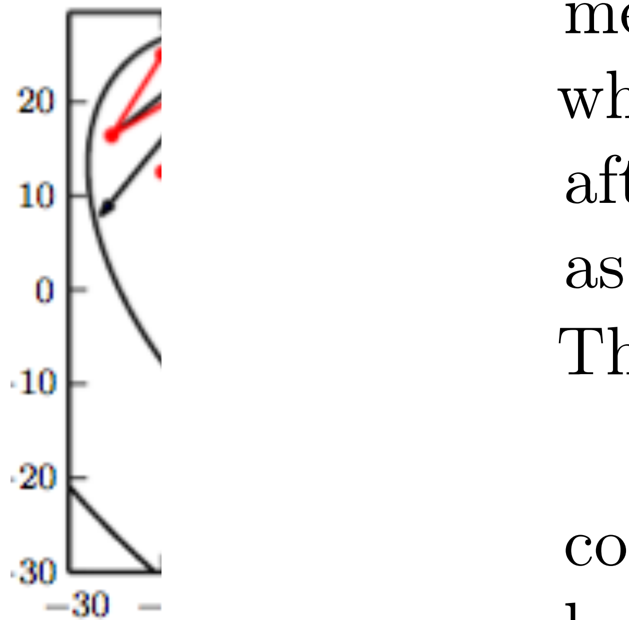
8.3.3 Nesterov Momentum

Sutskever *et al.* (2013) introduced a variant of the momentum algorithm inspired by Nesterov's accelerated gradient method (Nesterov, 1983, 20 update rules in this case are given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right],$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v},$$

where the parameters α and ϵ play a similar role as in the standard m



meth ... where ... after ... as at ... The c ... In ... conve ... by N

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

---

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

---

momentum does not improve the rate of convergence.

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov mom

SGD: Why Momentum Works?

Starting Point

Optimum

Solution

Step-size α = 0.0030

Momentum β = 0.0

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

https://distill.pub/2017/momentum/

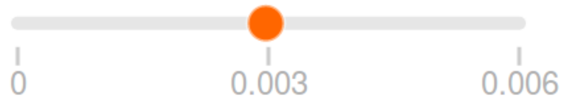
SGD: Why Momentum Works?

Starting Point

Optimum

Solution

Step-size α = 0.0030



Momentum β = 0.60

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

SGD: Why Momentum Works?

Starting Point

Optimum

Solution

Step-size α = 0.0030

Momentum β = 0.80

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

SGD: Why Momentum Works?

Starting Point

Optimum

Solution

Step-size α = 0.0030

Momentum β = 0.90

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

SGD: adaptive learning rates

Basic idea: **adapt the learning rate** for each parameter by taking into account an estimation of the curvature parameter-wise.

AdaGrad: good for convex problems. Rescales the learning rate inversely proportionally to the square root of the sums of the squares of all historical gradient values.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

SGD: adaptive learning rate

RMSProp: improved method for non-convex problems. Gradient accumulation is replaced by an exponentially weighted moving average.

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while

SGD: adaptive learning rate

Adam: integrates RMSProp with momentum, and uses a second-oder correction to correct the bias in the estimate of gradient and curvature, caused by exponential averaging. It is considered quite stable w.r.t. choice of parameters (not learning rate).

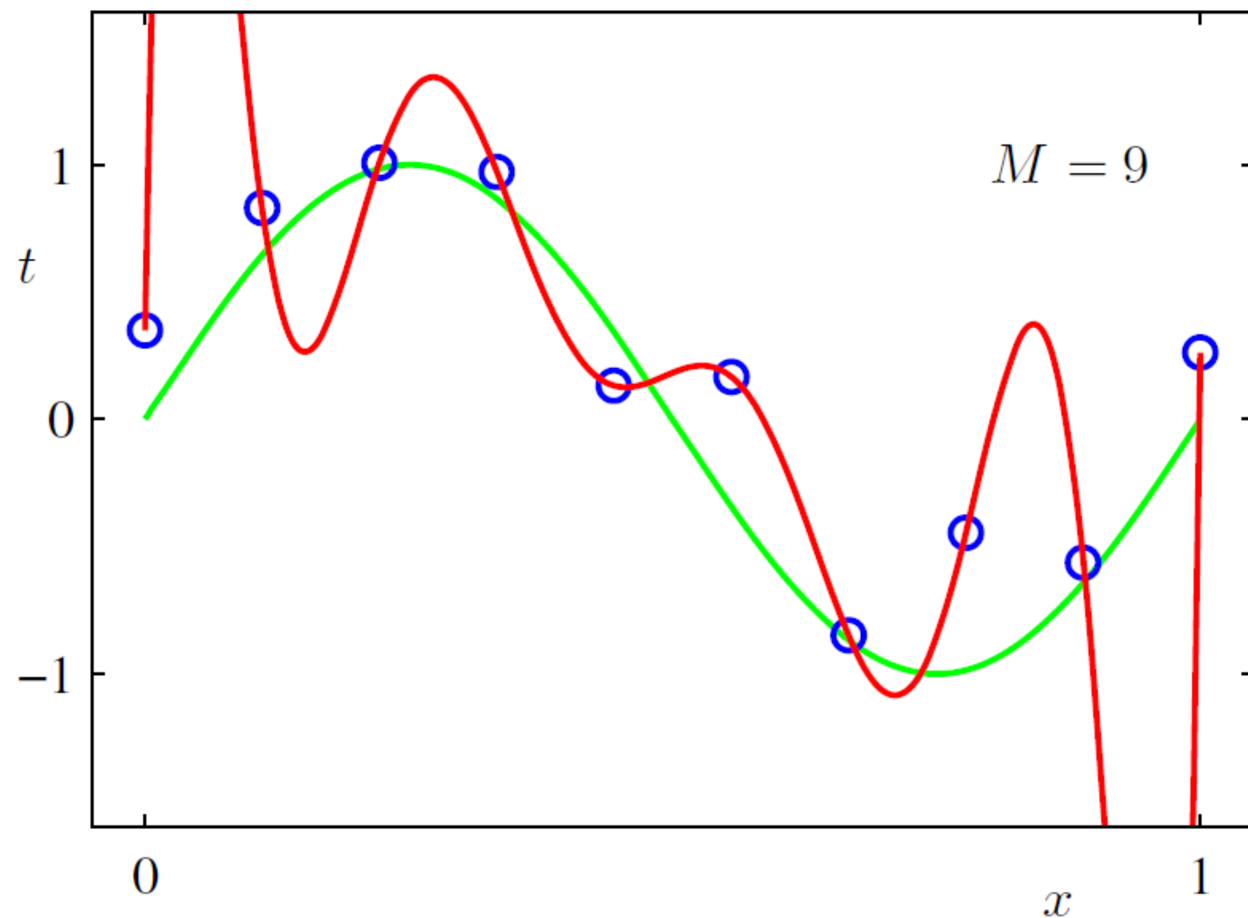
Algorithm 8.7 The Adam algorithm

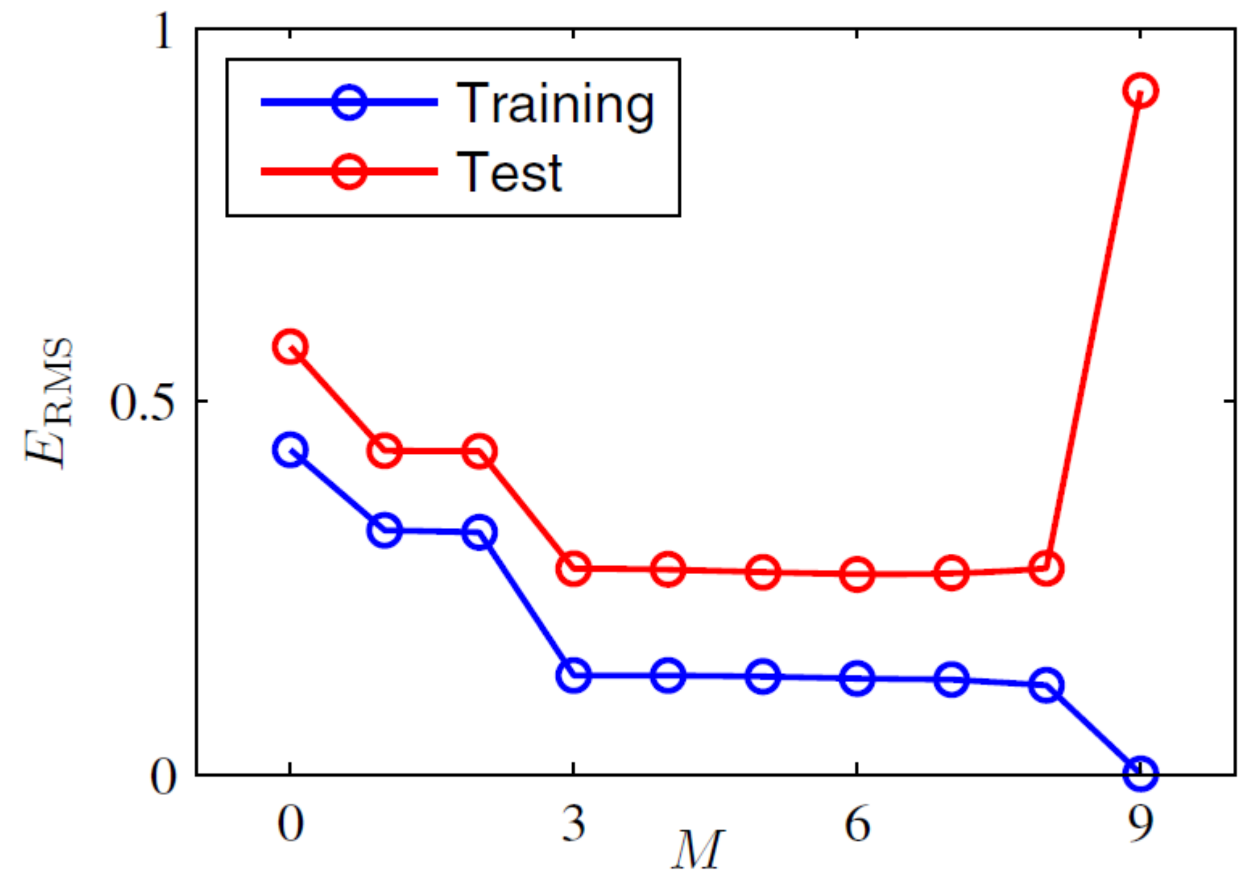
Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Regularisation of deep NN

Deep Neural Networks, with many layers and many nodes, typically have a very large model capacity. This can cause severe **overfitting**.

$t = \sin(2\pi x) + \epsilon$





Regularisation of deep NN

Deep Neural Networks, with many layers and many nodes, typically have a very large model capacity. This can cause severe overfitting.

Either there is a huge amount of data to train the model, or we need to **regularise** the learning algorithm.

Recommendation

use a model with very large capacity and regularise it well.

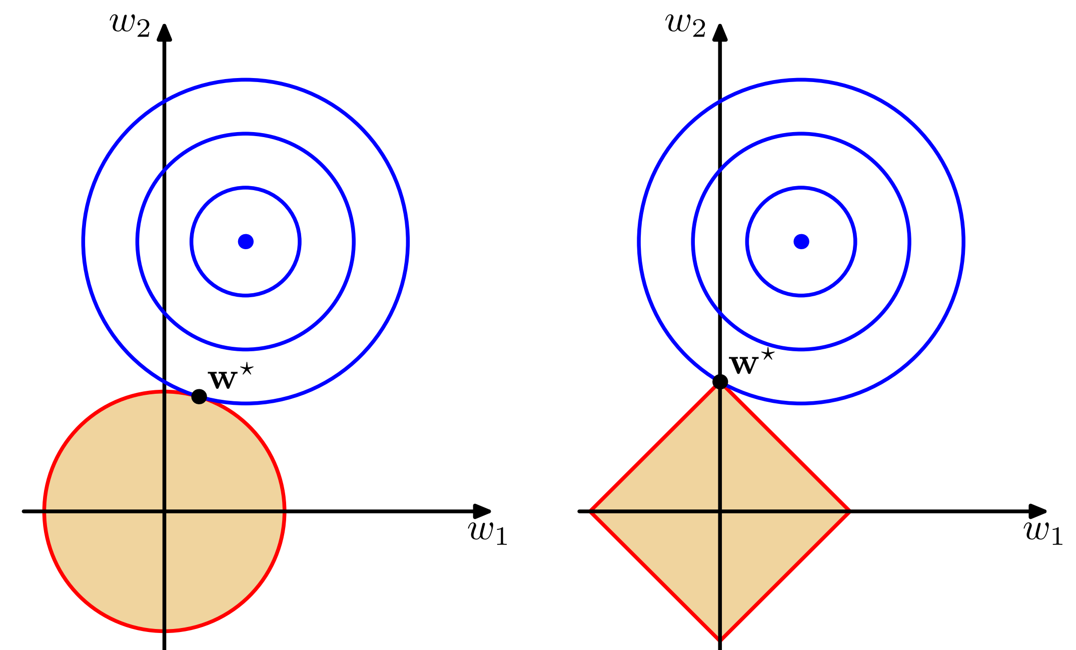
Classic regularisation:

* Typically operates on weights, in three different ways
* Hard constraint: $||\mathbf{w}||_p < r$
* Soft constraint: add a penalty term to cross entropy, depending on $||\mathbf{w}||_p$
* Bayesian view: place a prior on parameters and do MAP inference.

Regularisation of deep NN

Weight (soft) regularisation: add a penalty term to the likelihood/ cross entropy, penalising large weights.

- **L2 Norm**: $\Omega(\mathbf{w}) = \alpha||\mathbf{w}||_2^2$. Keeps weights small, corresponds to a Bayesian MAP with Gaussian prior. In NN: biases (constant terms) are not regularised, better to use one hyperparameter per layer.
- **L1 Norm**: $\Omega(\mathbf{w}) = \alpha||\mathbf{w}||_1$. Encourages sparsity of coefficients, corresponds to Bayesian MAP with Laplace prior.



Regularisation of deep NN

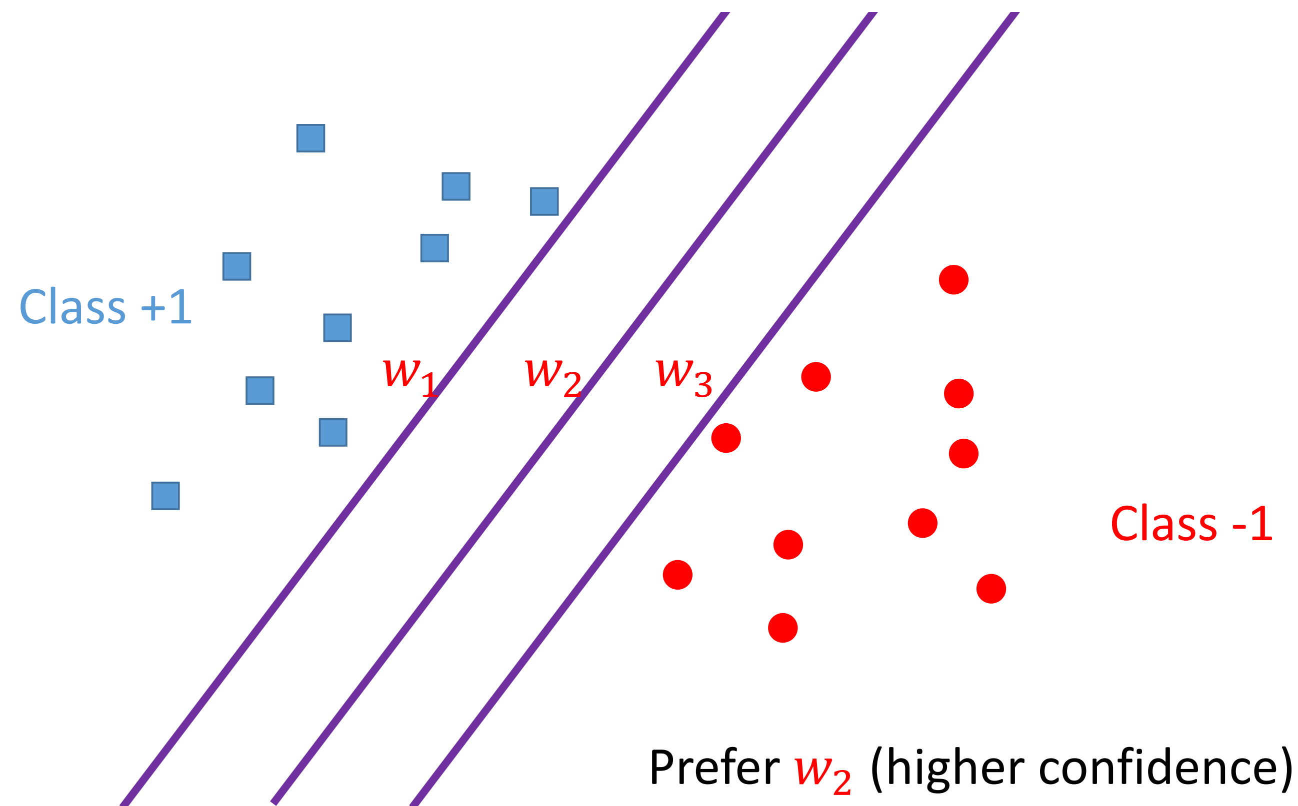
Noise-based regularisation: add noise to improve generalisation.

- **Augment dataset** by perturbed observations (good for classification problems with known invariance properties).
- **Perturb input points** to enhance robustness of learned solutions. One can also perturb weights or hidden layers (dropout).
- **Perturb weights** to enhance stability of learned solutions.

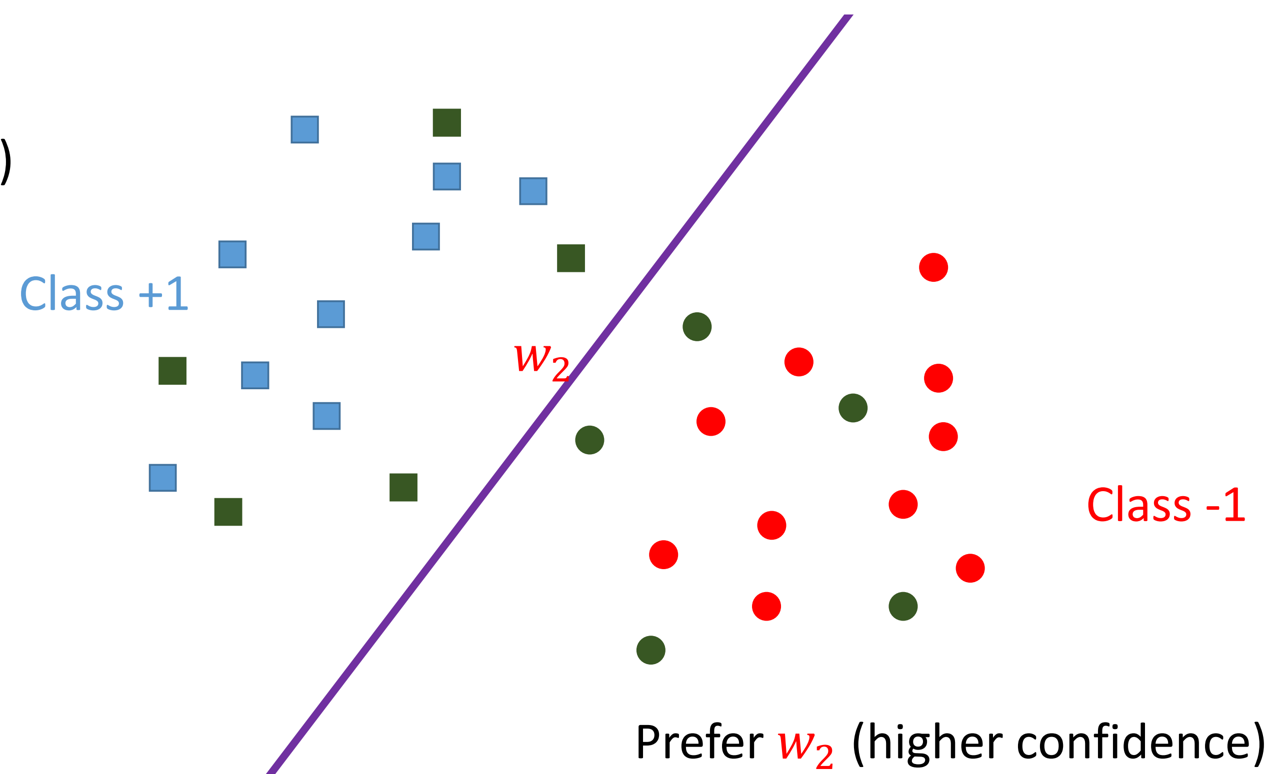
Other typical forms of regularisation are:

- **Early Stopping** which is the cheapest form of regularisation (self tuning hyperparameter)
- **Dropout** is a ensemble method inspired by bagging.
- **Gradient clipping** to avoid gradient explosion in SGD (skip connections help gradient vanishing).

Noise injection on inputs



Suppose there are multiple solutions, maybe because data are linearly separable.
Adding noise on inputs, makes the learning more stable.



But not too much noise, otherwise points may cross the boundary

Noise injection on inputs

Noise injection on inputs is equivalent to weight decay with L2 norm!

- Suppose the hypothesis is $f(x) = w^T x$, noise is $\epsilon \sim N(0, \lambda I)$
- After adding noise, the loss is

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f(x+\epsilon) - y]^2 = \mathbb{E}_{x,y,\epsilon}[f(x) + w^T \epsilon - y]^2$$

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f(x) - y]^2 + 2\mathbb{E}_{x,y,\epsilon}[w^T \epsilon (f(x) - y)] + \mathbb{E}_{x,y,\epsilon}[w^T \epsilon]^2$$

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f(x) - y]^2 + \lambda \left|\left|w\right|\right|^2$$

Noise injection on weights

- For the loss on each data point, add a noise term to the weights before computing the prediction

$$\epsilon \sim N(0, \eta I), w' = w + \epsilon$$

- Prediction: $f_{w'}(x)$ instead of $f_w(x)$
- Loss becomes

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f_{w+\epsilon}(x) - y]^2$$

Noise injection on weights

- Loss becomes

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f_{w+\epsilon}(x) - y]^2$$

- To simplify, use Taylor expansion
- $f_{w+\epsilon}(x) \approx f_w(x) + \epsilon^T \nabla f(x) + \frac{\epsilon^T \nabla^2 f(x) \epsilon}{2}$
- Plug in
- $L(f) \approx \mathbb{E}[f_w(x) - y]^2 + \eta \mathbb{E}[(f_w(x) - y) \nabla^2 f_w(x)] + \eta \mathbb{E}||\nabla f_w(x)||^2$

Small so can be ignored (applies to $\eta \mathbb{E}[(f_w(x) - y) \nabla^2 f_w(x)]$)

Regularization term (applies to $\eta \mathbb{E}||\nabla f_w(x)||^2$)

Noise injection on weights penalises functions varying too much locally. DNN are indeed affected by high local variability.

Data Augmentation

When we know predictions should be invariant to certain input transformations, we can generate many transformed copies of the input!

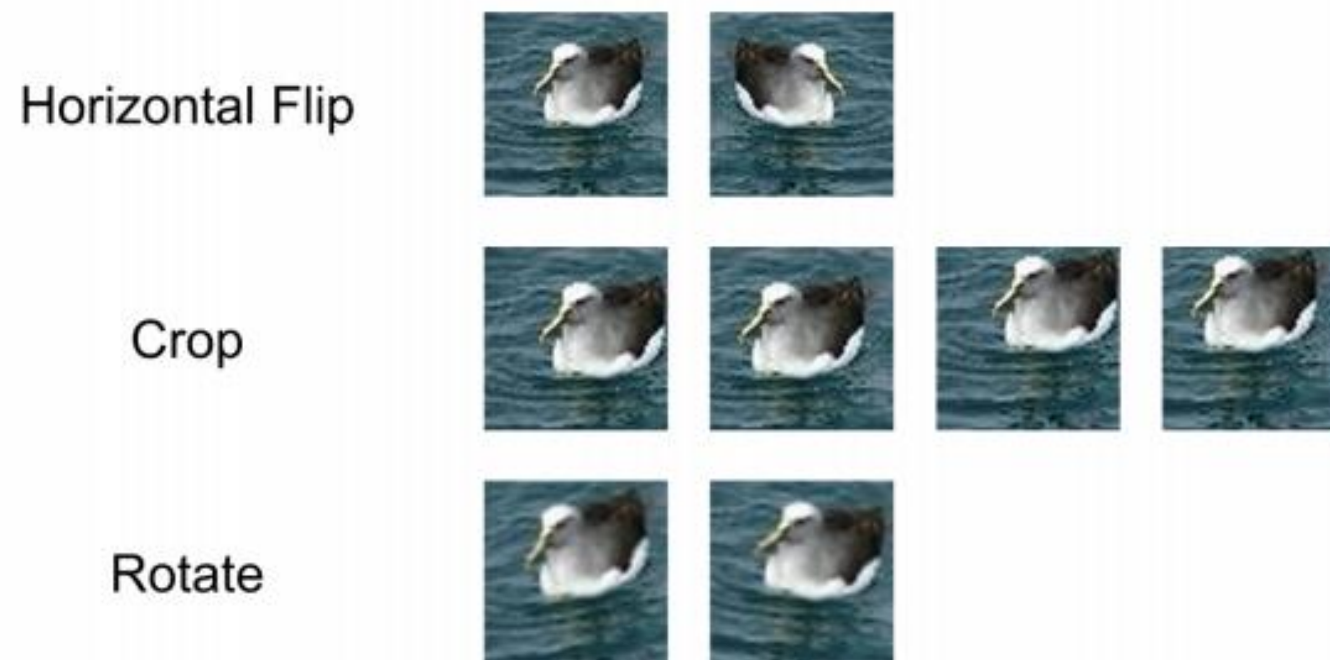


Figure from *Image Classification with Pyramid Representation and Rotated Data Augmentation on Torch 7,* by Keven Wang

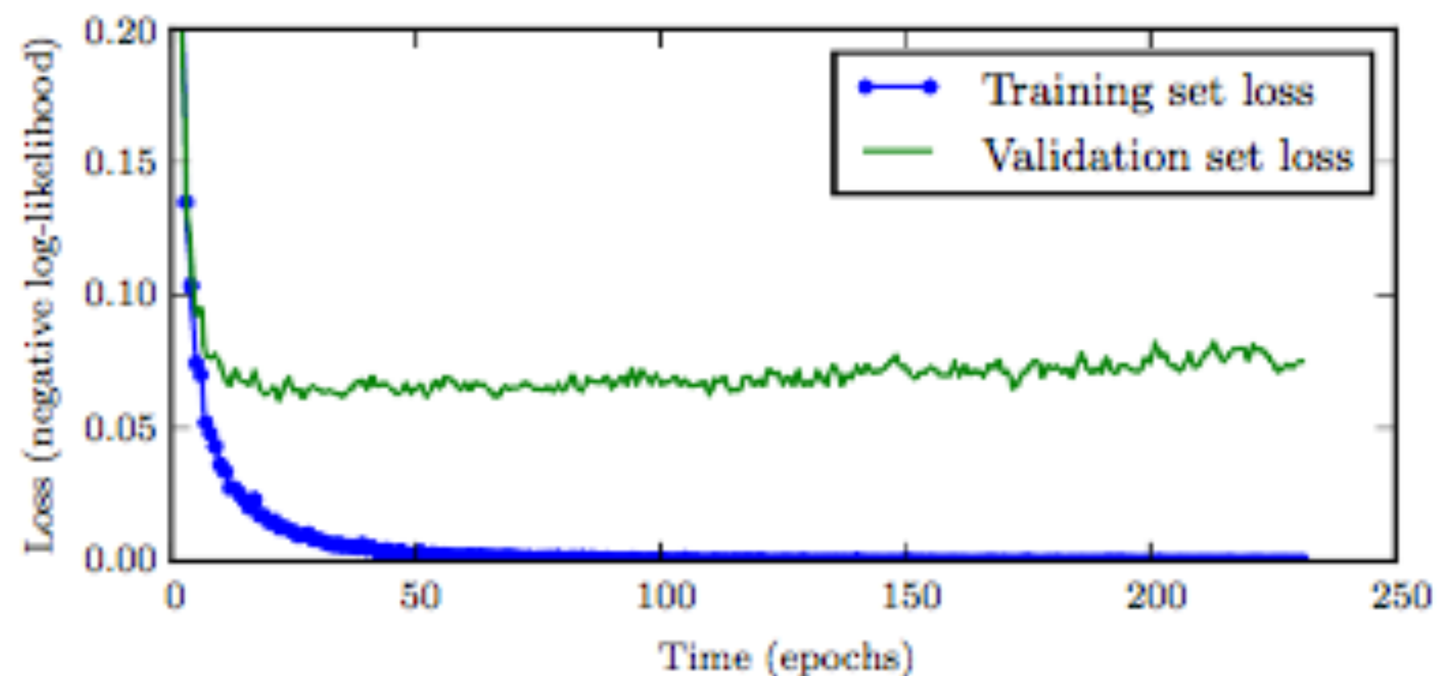
We need to be careful, as some transformations may not be invariant (e.g. rotation of 180 degrees for handwritten digits.

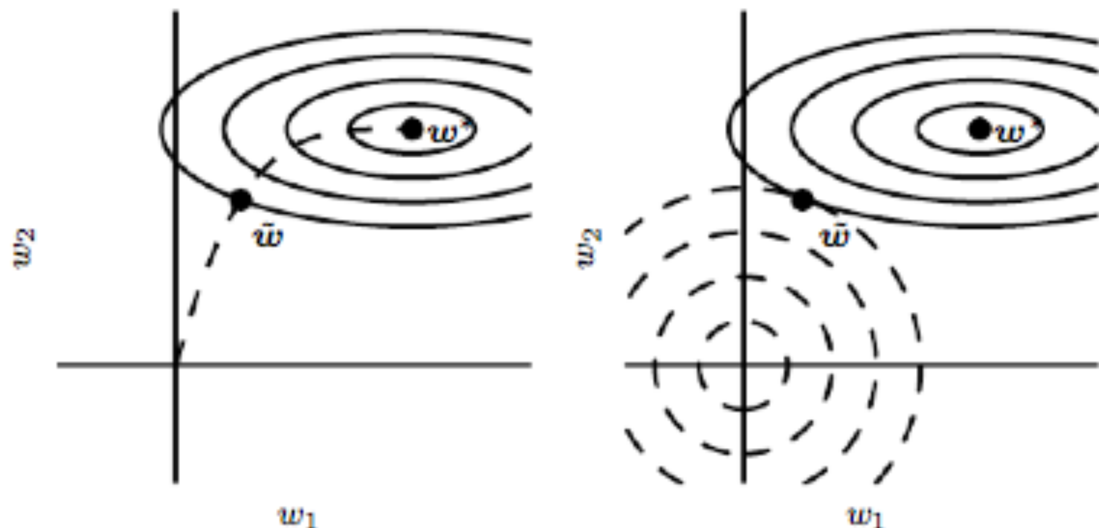
Early Stopping

One of the most common regularisation strategies. It is very simple but effective and computationally cheap.

Idea: a good generalisation is not found in a local optimum of cross entropy, but typically somewhere in the path to it.

Practically: monitor a validation dataset while running SGD, and stop when the validation error starts to increase.
Number of steps of SGD becomes an hyperparameter.





Intuition: Early stopping (for linear regression) **corresponds to L2 regularisation**, but learns the optimal weight decay hyperpar on the fly. Relationship: $\tau \approx 1/\epsilon\alpha$, where ϵ depends on the eigenvalues of Hessian at minimum.

Early Stopping

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*

Early Stopping

Validation data costs in terms of dataset size. To improve, one can use early stopping to identify the number of iterations, and retrain the model with the whole data.

- How to reuse validation data
  1. Start fresh, train with both training data and validation data up to the previous number of epochs
  2. Start from the weights in the first run, train with both training data and validation data util the validation loss < the training loss at the early stopping point

Bagging

Bagging (bootstrap aggregation) is an **ensemble method/ model averaging** strategy: different models are trained on the same dataset and their average is returned.

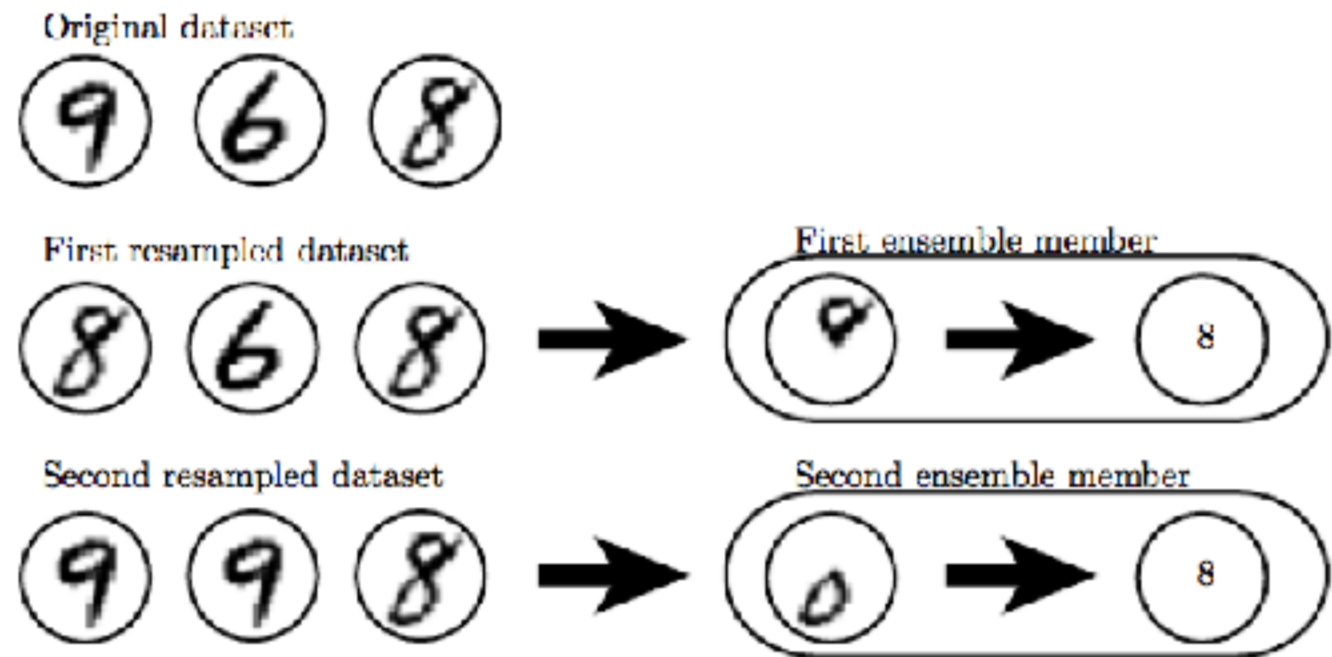
Idea: different models typically do different (**independent**) **errors** on predictions.

Bagging typically trains the same model on different datasets generated by sampling with repetition, as in bootstrapping.

Suppose we have k models, each with error ϵ_i, with variance $\mathbb{E}[\epsilon_i^2]=v$ and covariance $\mathbb{E}[\epsilon_i \epsilon_j]=c$. Then by averaging models, we average errors. The expected squared error is then:

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j\right)\right]$$

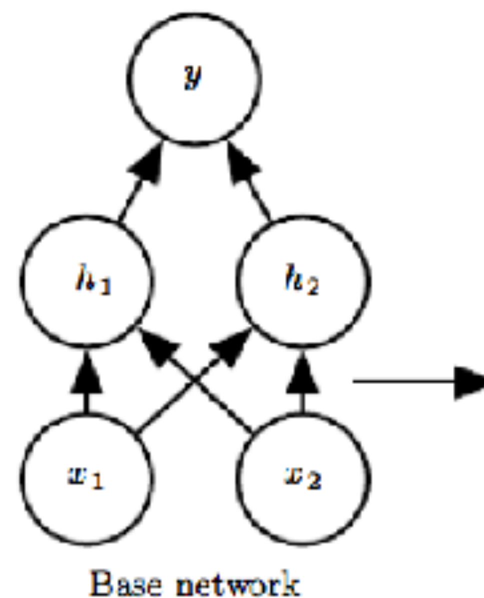
$$= \frac{1}{k}v + \frac{k-1}{k}c.$$



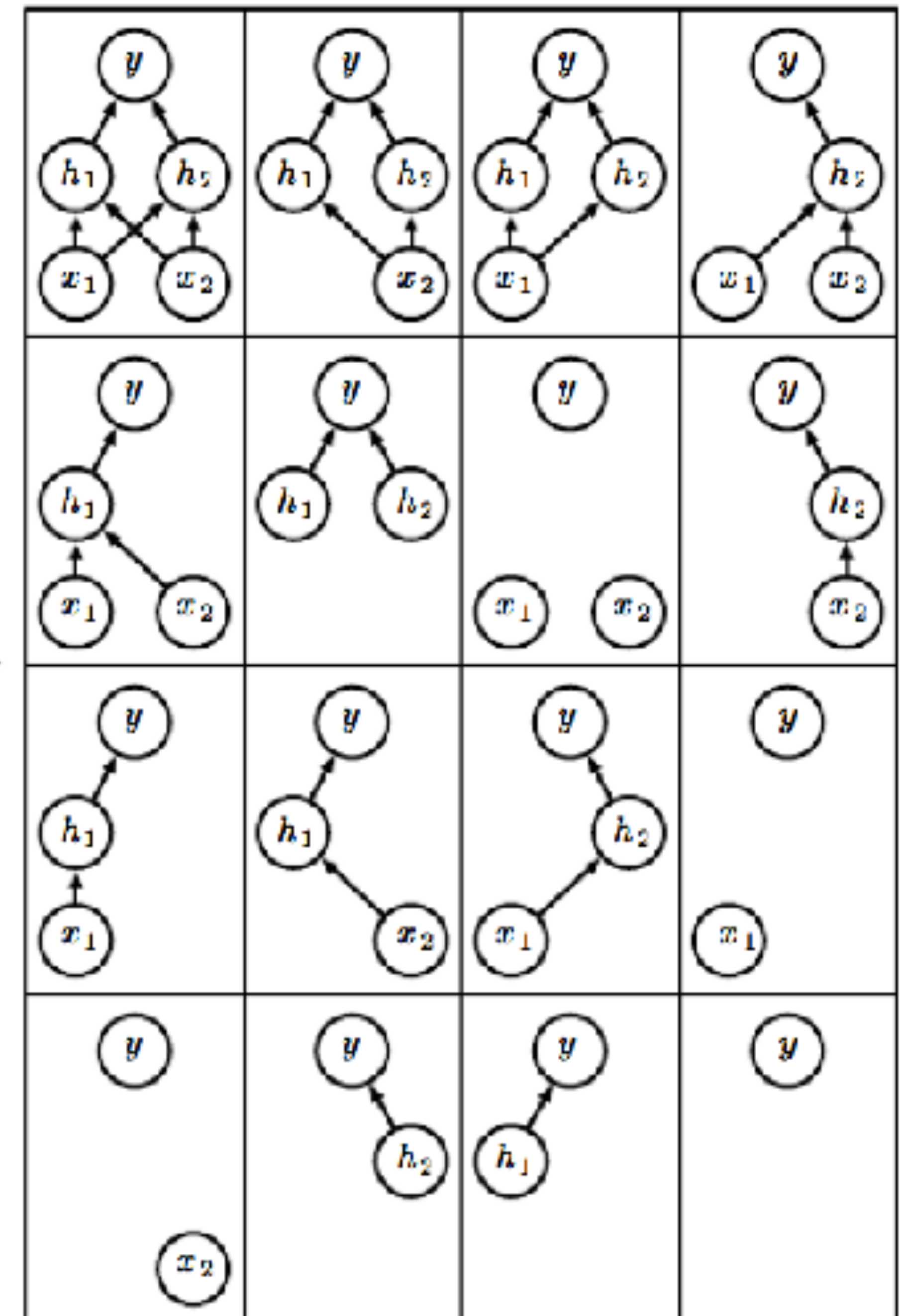
Dropout

Dropout is a form of bagging, in which we consider all models obtained by removing all possible subset of the hidden and input nodes. If there are m such nodes, there are 2^m c

Dropout simultaneously train all such modes, sharing their parameters, by sampling a **mask** μ each time a new training point is processed by SGD: each node is kept with probability p (typically 0.5 for hidden, 0.8 for input).

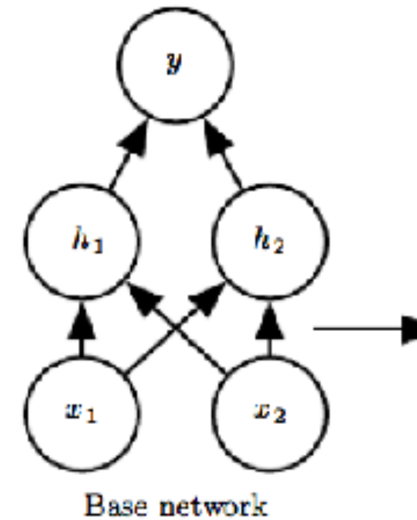


Base network

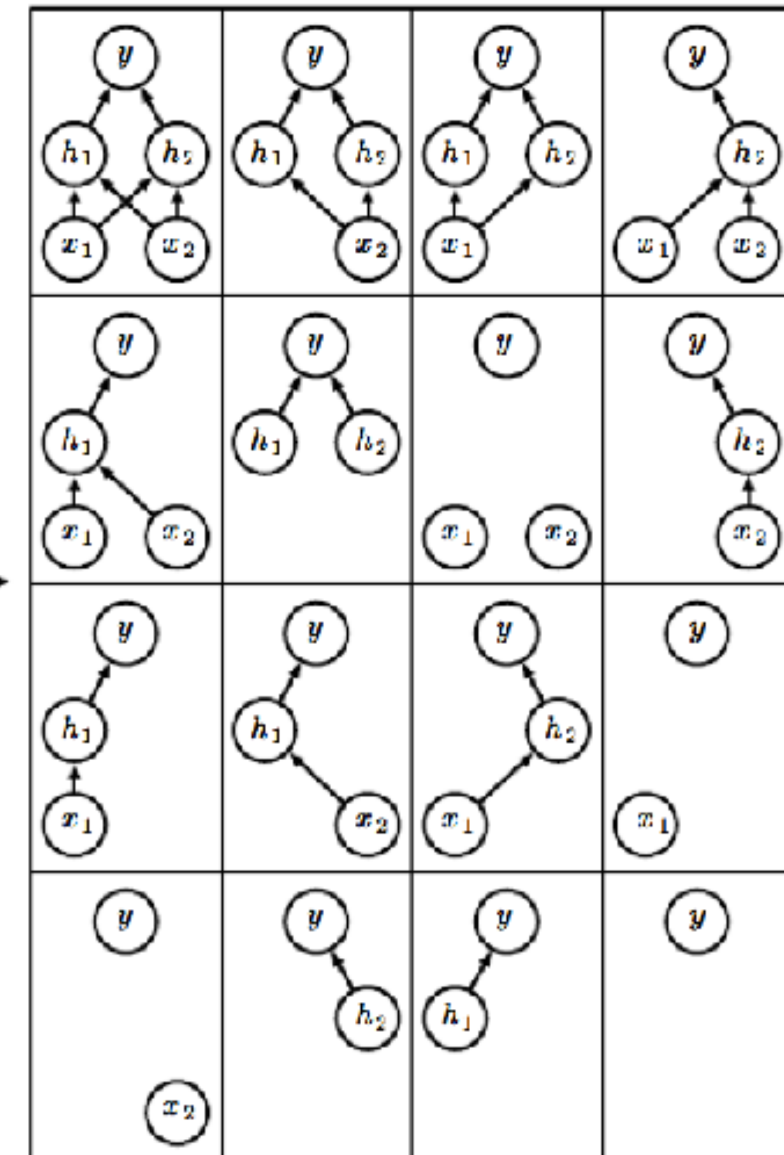

Ensemble of subnetworks

Dropout

Inference is performed either by sampling few masks and averaging, or by the **weight scaling approximation**: each node is multiplied by the masking probability p during inference.



Base network


Ensemble of subnetworks

Dropout can be seen as an intelligent perturbation of input data (by erasing/ perturbing features rather than inputs).

Sparse Representation and Batch Normalisation

Sparse representation: add a term penalising large values of hidden units after applying the activation function. Using L1 norm bring sparsity on active hidden units.

Batch normalisation: standardise hidden units during learning, on a minibatch of training points.
This has the effect of stabilising the learning algorithm on a deep architecture, by fixing the distribution of each node in each layer, and allows the use of larger learning rates.

[known phenomenon: **covariate shift** – change in distribution of inputs of a layer makes training unstable.]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1 \ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$