# Regularization in Deep Learning

Jann Goschenhofer

LMU - Institute of Statistics
Seminar "Introduction to Deep Learning"

*jann.goschenhofer@posteo.de*

January 13, 2017

# Structure

# What is regularization?

*Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error*

[Goodfellow et al., 2016]

# Concept of overfitting

- Goal of model: perform well on unobserved input data
- Two competing tasks:

↓Train error vs. ↓Gap between train and test ($\widehat{=}$ generalization) error
↓Underfitting vs. ↓Overfitting

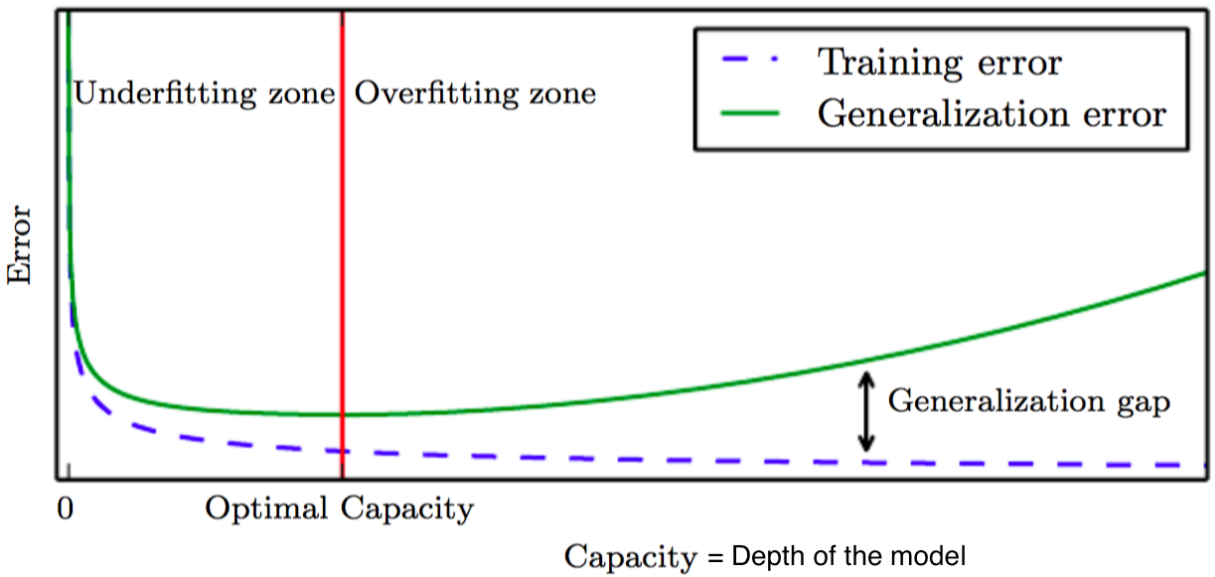



**Figure:** [Goodfellow et al., 2016]

- Generalization gap depends on ratio: $\frac{\text{complexity of the model}}{\#\text{train data}} \widehat{=} \frac{\text{depth of the net}}{\#\text{train data}}$

# Illustration of overfitting

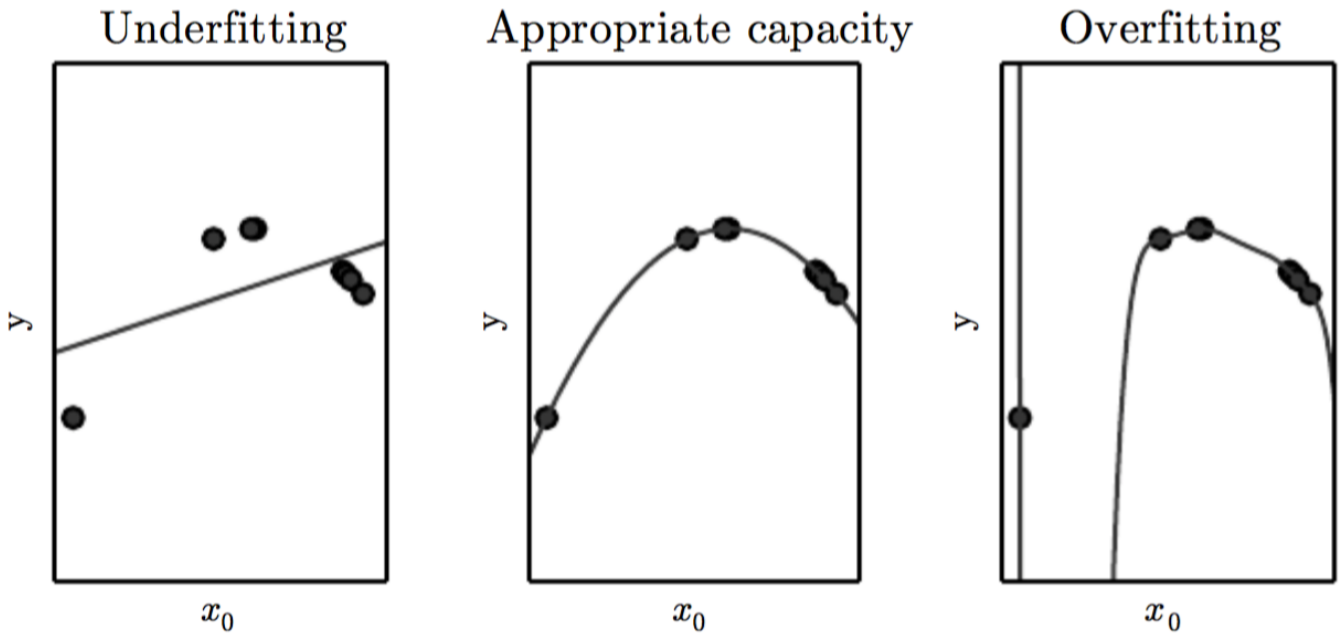



**Figure:** [Goodfellow et al., 2016]

with capacity $\widehat{=}$ model complexity $\widehat{=}$ depth of net $\widehat{=}$ number of updates $m$

# Structure

# Basic concept

- Limit capacity of model $\rightarrow$ limit expression of its parameters
- Penalize the model for large parameters
- Add penalty $\Omega(\theta)$ on the parameter norm such that:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta), \text{ with } \alpha \geq 0 \tag{1}$$

- Linear models: parameters $\theta \mathrel{\widehat{=}}$ coefficients $\beta$
- Neural networks: parameters $\theta \mathrel{\widehat{=}}$ weights $\omega$
- No penalization of biases $b$:
  - Less data required for fitting $b$ than $\omega$
  - One bias per unit, n weights per unit (in vanilla net)
  - Possible consequence of bias-regularization: underfitting

# $L^2$-penalization // the superstar

- Analogue to **ridge regression**: $\Omega(\theta) = \frac{1}{2}||\omega||_2^2$ such that

$$\tilde{J}(\theta; X, y) = \alpha\omega^T\omega + J(\theta; X, y) \tag{2}$$

with gradient:

$$\nabla_\omega \tilde{J}(\theta; X, y) = \alpha\omega + \nabla_\omega J(\theta; X, y) \tag{3}$$

- Most popular regularizer
- One weight update using gradient descent:

$$\begin{aligned} &\omega^{m+1} \leftarrow \omega^m - \gamma(\nabla_\omega \tilde{J}(\theta; X, y)) = (1 - \gamma\alpha)\omega^m - \gamma\nabla_\omega J(\omega; X, y) \\ &\Rightarrow \text{Shrinkage of } \omega \text{ in each iteration} \end{aligned} \tag{4}$$

- Therefore termed **weight decay** in neural net applications

# Discovering the effect of weight decay I

1. Quadratic Taylor-approximation of unregularized $J(\theta)$ with $\theta = \omega$ at minimum $\omega^* = \mathit{argmin}_\omega J(\omega)$:

$$\begin{aligned}\hat{J}(\theta) &= J(\omega^*) + \frac{1}{1!}\frac{\partial J(\omega^*)}{\partial \omega}(\omega - \omega^*) + \frac{1}{2!}\frac{\partial^2 J(\omega^*)}{\partial \omega \partial \omega}(\omega - \omega^*)^2 \\ &= J(\omega^*) + \frac{1}{2}(\omega - \omega^*)^T H(\omega - \omega^*)\end{aligned} \tag{5}$$

   with $H$ being the positive-semidefinite Hessian of J at $\omega^*$
2. The minimum of $\hat{J}$ occurs where its first derivatives equal 0:

$$\nabla_\omega \hat{J}(\omega) = H(\omega - \omega^*) = 0 \tag{6}$$

3. Transform into regularized version:

$$\begin{aligned}H(\tilde{\omega} - \omega^*) + \alpha\tilde{\omega} = 0 \Leftrightarrow (H + \alpha I)\tilde{\omega} = H\omega^* \\ \tilde{\omega} = (H + \alpha I)^{-1} H\omega^*\end{aligned} \tag{7}$$

# Discovering the effect of weight decay II

4. Apply eigendecomposition as H is real and symmetric, such that $H = Q\Lambda Q^T$:

$$\begin{aligned} \tilde{\omega} &= Q(\Lambda + \alpha I)^{-1}\Lambda Q^T \omega^* \\ &\Rightarrow \text{collapses to } \omega^* \text{ for } \alpha = 0 \end{aligned} \tag{8}$$

   Where
   - $\Lambda = diag(\lambda_1, ..., \lambda_d)$ with $\lambda_i \widehat{=}$ eigenvalue i of $H$
   - $Q \widehat{=}$ orthonormal basis of eigenvectors $v_i, i = 1, ..., d$

5. Effect on weight $i$:

$$\begin{aligned} \tilde{\omega}_i &= v_i \frac{\lambda_i}{\lambda_i + \alpha} v_i^T \omega_i^* \overset{v_i v_i^T = 1}{=} \frac{\lambda_i}{\lambda_i + \alpha} \omega_i^* \\ &\Rightarrow \textbf{ rescaling of } \omega^* \text{ with eigenvalues of the Hessian} \end{aligned} \tag{9}$$

# Discovering the effect of weight decay III

6. $\lambda_i$ resembles sensitivity of $J$ on $\omega_i$ // importance of $\omega_i$
   - $\tilde{\omega}_i = \frac{\lambda_i}{\lambda_i+\alpha}\omega_i^*$
   - $\lambda_i \gg \alpha$ : high $\lambda_i \rightarrow$ high curvature $\rightarrow$ important weight $\rightarrow$ low shrinkage of update
   - $\lambda_i \ll \alpha$ : small $\lambda_i \rightarrow$ low curvature $\rightarrow \omega_i^m$ "unnecessary" weight $\rightarrow$ heavy shrinkage

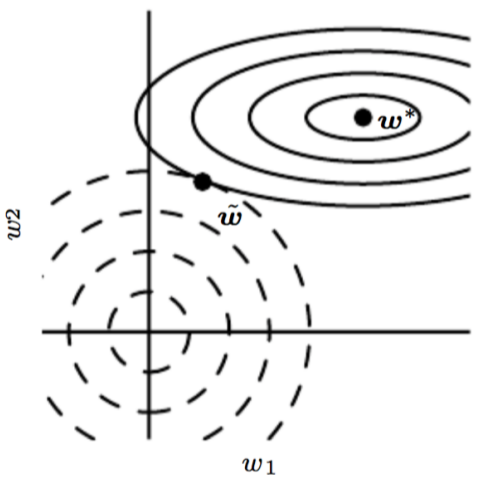



**Figure:** [Goodfellow et al., 2016]

# $L^1$-penalization // the eradicator

- Analogue to **lasso regression**: $\Omega(\theta) = ||\omega||_1$ such that

$$\tilde{J}(\theta; X, y) = \alpha||\omega||_1 + J(\theta; X, y) \tag{10}$$

leading to gradient:

$$\nabla_\omega \tilde{J}(\theta; X, y) = \alpha sign(\omega) + \nabla_\omega J(\theta; X, y) \tag{11}$$

- Harder penalization: shrinks weights **and** sets some to 0
- Creates **sparse** models and is therefore used for **feature selection**

# Understanding $L^1$-penalization I

- Taylor-approximation 2.0 of $\hat{J}(\theta)$
  - Assumption: uncorrelated inputs $\rightarrow H = diag(H_{1,1}, ..., H_{n,n}), H_{i,i} > 0$
  - $L^1$ regularized objective function:

$$\hat{J}(\theta; X, y) = J(\omega^*; X, y) + \sum_i \left[ \frac{1}{2} H_{i,i} (\omega_i - \omega_i^*)^2 + \alpha |\omega_i| \right] \tag{12}$$

- Solution of the minimization problem:

$$\tilde{\omega}_i = sign(\omega_i^*) max \left[ |\omega_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right] \tag{13}$$

with two possible situations given $\omega_i^* > 0$:

  - High $H_{i,i} \rightarrow$ high curvature $\rightarrow$ low penalization of "necessary" weight $\rightarrow$ weight is just shrinked $\rightarrow 0 < \tilde{\omega}_i < \omega_i^*$
  - Low $H_{i,i} \rightarrow$ low curvature $\rightarrow$ hard penalization of "unnecessary" weight $\rightarrow \omega_i^* \leq \frac{\alpha}{H_{i,i}} \rightarrow \tilde{\omega}_i = 0$

# Understanding $L^1$-penalization II

- Compare both methods with $\lambda_i = H_{i,i}$ and $\omega_i^* > 0$:

$$\tilde{\omega}_i^{L^2} = \frac{H_{i,i}}{H_{i,i} + \alpha}\omega_i^* \quad \text{vs.} \quad \tilde{\omega}_i^{L^1} = max\left[|\omega_i^*| - \frac{\alpha}{H_{i,i}}, 0\right] \tag{14}$$

**$L^2$-penalization solely shrinks weights towards 0 and $L^1$-penalization furthermore eradicates weights**

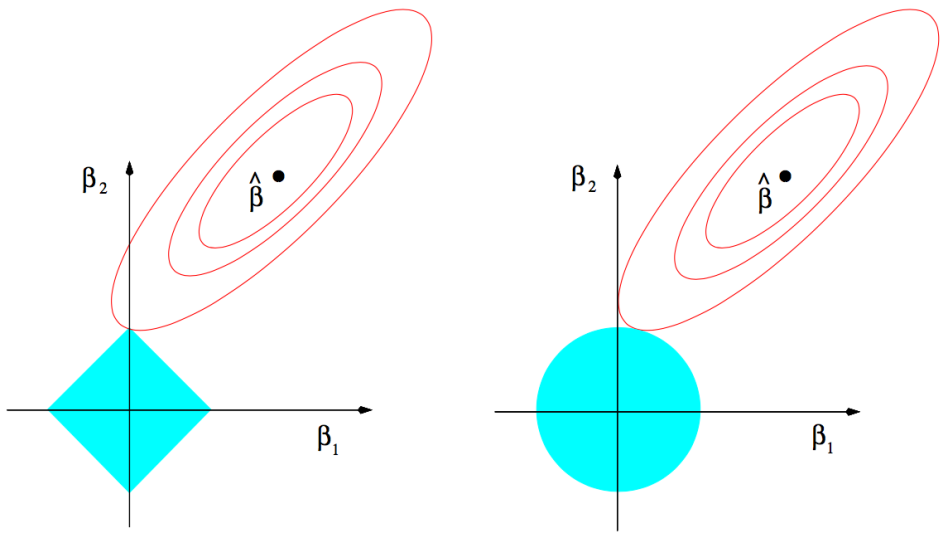



**Figure:** [Hastie et al., 2009]

# Max-norm-penalization // the underdog

- Direct approach to restrict weights
- Absolute upper bound for norm of incoming weight vector at each neuron:

$$||\omega_j||_2 \leq c \text{ for } \textit{neuron}_j \text{ where } \omega_j = (\omega_{1,j}, ..., \omega_{n,j})^T$$

- Empirical finding: $c \approx 4$
- Originated in matrix factorization applications, not in machine learning [Shen et al., 2014]
- Gained popularity as it performs well in combination with **dropout**
- R Implementation of all methods in `h2o.deeplearning()` from package `h2o`. In `caret` solely implementation of weight decay

# Structure

# Basic concept

- Recap: low generalization because high ratio $\frac{\text{complexity of the model}}{\#\text{train data}}$
- Increase train data $\rightarrow$ improve robustness $\rightarrow$ better generalization
- Limited data supply $\rightarrow$ create fake data
- Increase variation in inputs **without** changing the labels
- Link to **Noise Injection**

# Application

- Image and Object recognition:
  Rotation / Scaling / Pixel translation / Flipping / Noise Injection / Vignetting / Color Casting / Lens Distortion / Injection of random negatives
  - R implementation in `imager` and `OpenImageR`
  - See [Wu et al., 2015] for deeper insights
- Speech recognition:
  Speed augmentation / Vocal Tract Perturbation [Cui et al., 2015]

# Graphic Examples

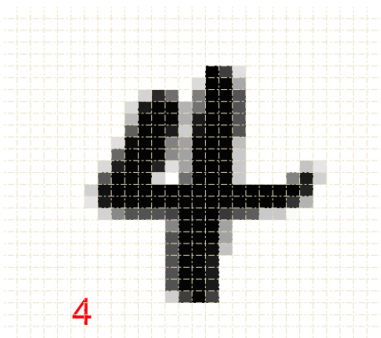
**(a)** Original

**(b)** Color

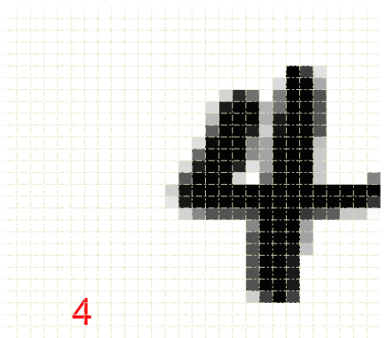
**(c)** Rotate

**(d)** Horizontal Stretch

**Figure:** [Wu et al., 2015]




**(a)** Original




**(b)** Moved

# Structure

# Basic concept

- Goal: find optimal number of iterations $m^*$
- Stop algorithm early // before generalization error increases

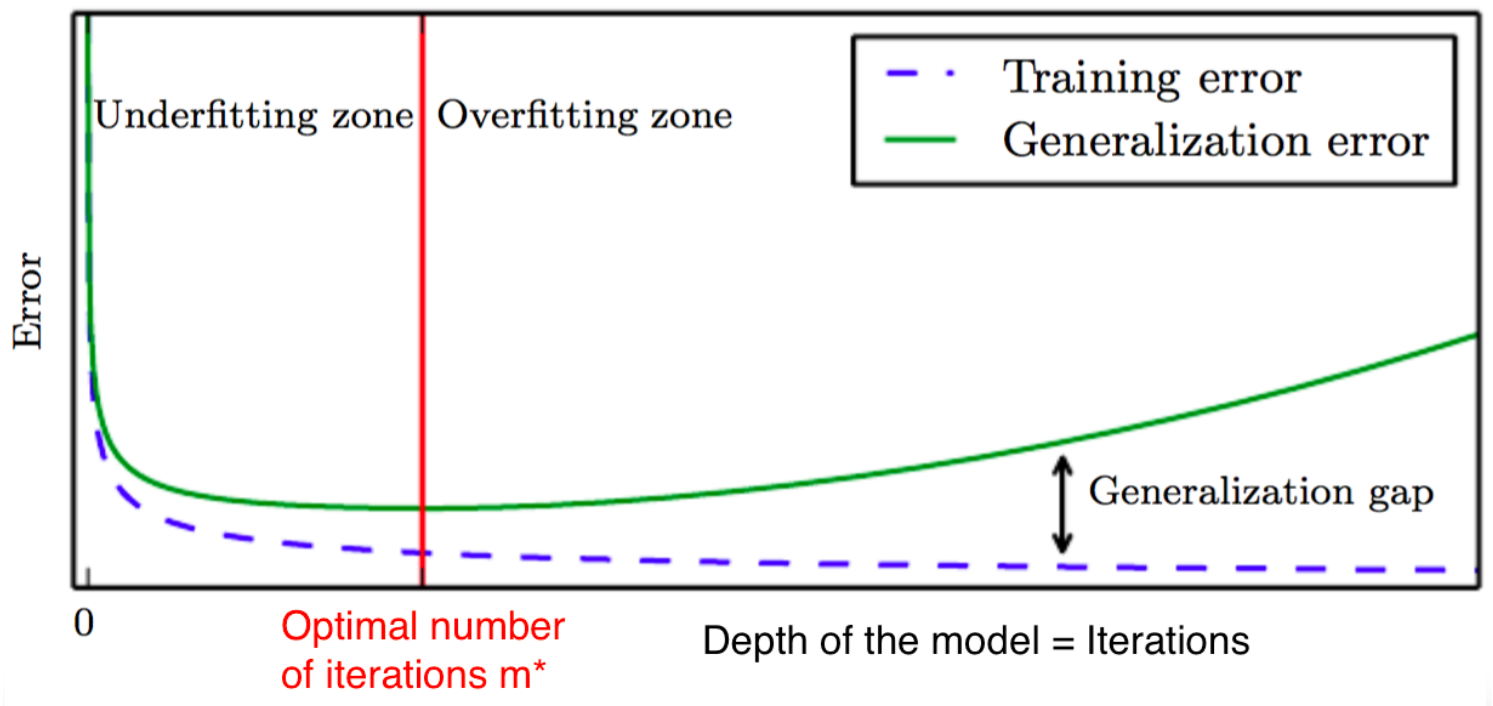



**Figure:** [Goodfellow et al., 2016]

# How to early stopping

- Split train data: $data^{train} = data^{subtrain} + data^{validation}$
- Monitor the error rate on the $data^{validation}$ in steps $n$ and stop the algorithm when it stops decreasing
- Store copy of model parameters after each error-improving step
- Use them as final parameters when there is no improvement after a range of steps $p$ ("patience")
- R implementation: `h2o.deeplearning` with option `stopping_rounds`, disabled in `mlr`

# Strengths and weaknesses

| Strengths | Weaknesses |
|---|---|
| Effective and simple | Periodical evaluation of $data^{validation}$-error |
| Applicable to almost any model without adjustment | Temporary copy of $\theta^*$ |
| Combinable with other regularization methods | Less data for training $\rightarrow$ include $data^{validation}$ afterwards:<br>1. determine $m^*$, retrain with whole $data^{train}$<br>2. determine $\theta^*$, continue training with $data^{train}$ while $\text{Error}(\theta, data^{train}) < \text{Error}(\theta^*, data^{validation})$ |

# Early stopping as weight decay I

- [Goodfellow et al., 2016] proof (under some assumptions) this relation of early stopping's optimal iterations $m$ and weight decay's penalization parameter $\alpha$:

$$m \approx \frac{1}{\gamma\alpha} \Leftrightarrow \alpha \approx \frac{1}{m\gamma}$$

- Small $\alpha \mathrel{\widehat{=}}$ low penalization $\Rightarrow$ high $m \mathrel{\widehat{=}}$ deep model
- Recap weight decay: weights corresponding to directions of high curvature (high $\lambda_i$) are less regularized by weight decay than directions of low curvature (low $\lambda_i$)
- Weights from the first case learn early relative to the latter

# Early stopping as weight decay II

- Graphic interpretation:

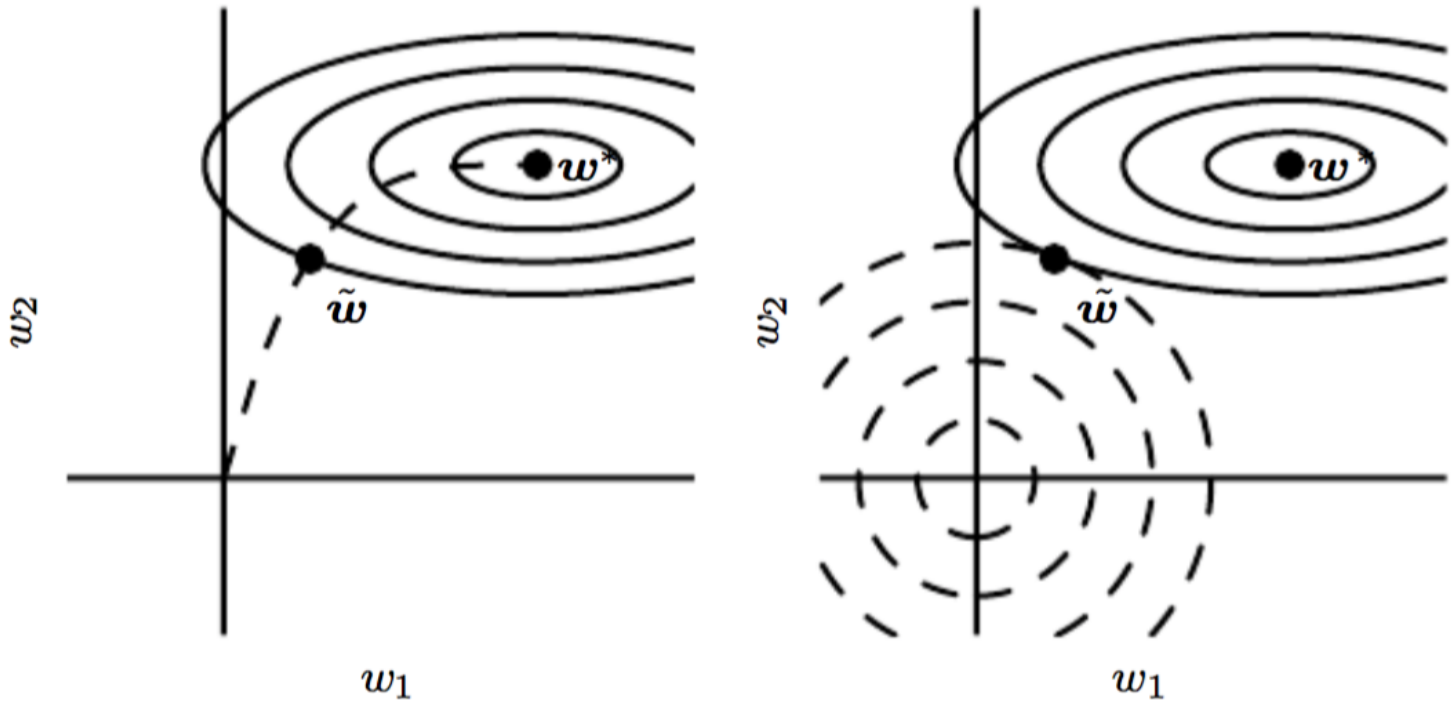



**Figure:** Optimization path in early stopping (left) and weight decy (right) after [Goodfellow et al., 2016]

# Structure

# Basic concept

- Combining several models trained on same task $\rightarrow$ reduce generalization error
- Vote of **ensemble** of models is then averaged for prediction
- Different models will make different errors
- Shortcomings of one ensemble member balanced by the **comittee**
- Methods differ in the way the ensemble is constructed
- Popular examples: **Bagging** and **Boosting**

# Illustration Ensemble Learning

Assume: $k$ models with $\epsilon_i \sim N(0, v)$, $\mathbb{E}[\epsilon_i, \epsilon_j] = c$ and $\mathbb{E}[\epsilon_{ensemble}] = \frac{1}{k}\sum_{i=1}^{k} \epsilon_i$

$$
\begin{aligned}
MSE_{ensemble} &= \mathbb{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i\right)^2\right] = \frac{1}{k^2}\sum_{i=1}^{k}\left[\mathbb{E}[\epsilon_i^2] + \sum_{i\neq j}\mathbb{E}\left[\epsilon_i, \epsilon_j\right]\right] \\
&= \frac{1}{k^2}\sum_{i=1}^{k}\left[\mathbb{E}[\epsilon_i^2] + (k-1)\mathbb{E}\left[\epsilon_i, \epsilon_j\right]\right] = \frac{1}{k^2}k\left[v + (k-1)c\right] \\
&= \frac{1}{k}v + \frac{k-1}{k}c
\end{aligned}
\tag{15}
$$

- Perfectly correlated models: $c = v \rightarrow MSE_{ensemble|c=v} = v$
- Perfectly uncorrelated models: $c = 0 \rightarrow MSE_{ensemble|c=0} = \frac{1}{k}v$

# Illustration Bagging (bootstrap aggregation)

- Construct $k$ learners on $k$ data subsets drawn with replacement from original train data set
- Here: "digit 8 detector"

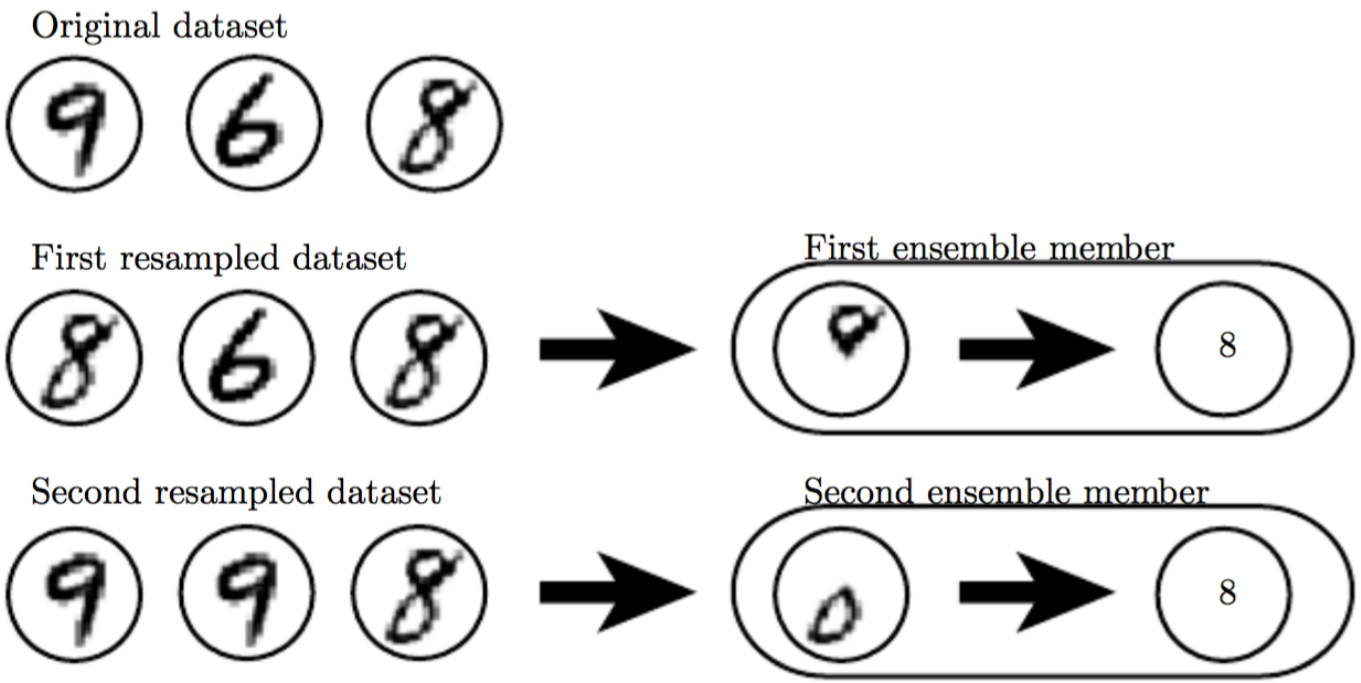



**Figure:** [Goodfellow et al., 2016]

# Structure

# Basic concept

- Problem: overfitting, low generalization
- Idea: force net to generalize by randomly dropping units during training
- Units hindered from relying on each other
- Train ensemble of "thinned sub-nets" and use their collective vote for prediction
- Extensive parameter sharing amongst sub-nets
- Each sub-net solely trained on single train data point

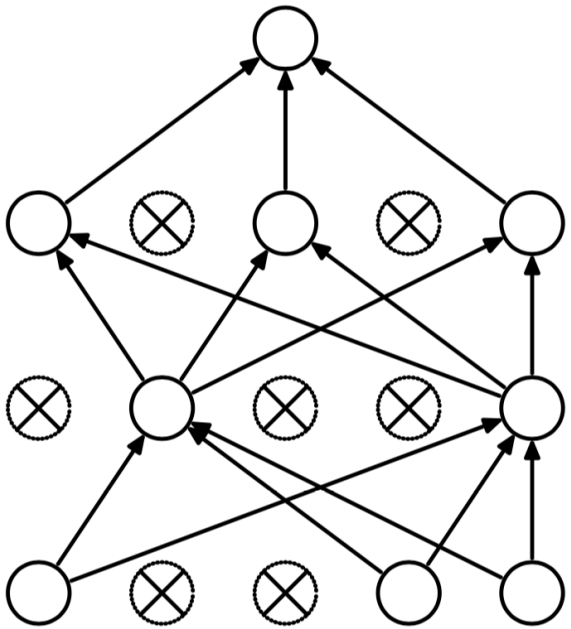


**Figure:**
[Srivastava et al., 2014]

# What happens internally?

- Sample from $2^d$ possible models that share parameters for computational feasibility where $d \widehat{=} \#$ units
- Sub-net with objective function $J(\theta, \mu)$ depends on mask $\mu$ that randomly determines in/exclusion of units and is trained on one randomly sampled train data point $(y^{(i)}, X^{(i)})$
- Mask $\mu$ is vector of length $d$ with $\mu = (\mu_1, ..., \mu_d)$, $\mu_l = \{0; 1\}$ and $P(\mu_l = 1) = p$
- Weights of sub-nets are shared through iterations

# Recap: train net using stochastic gradient descent

- $m \widehat{=}$ # iterations $\widehat{=}$ # data points from shuffled train sample
- $K \widehat{=}$ # parameters $\theta_j$ to be estimated
- $\gamma \widehat{=}$ learning rate of the algorithm
- $J(\theta)_i \widehat{=}$ objective function for iteration $i$

**Algorithm 1** stochastic gradient descent update

1: randomly shuffle train sample
2: **repeat**
3: **for** $i = 1, ..., m$ **do**
4: **for** $j = 1, .., K$ **do**
5: $\theta_j^{(i+1)} = \theta_j^{(i)} - \gamma \frac{\partial J(\theta)_i}{\partial \theta_j}$
6: **end for**
7: **end for**
8: **until** stop criterion is reached

# Train net using stochastic gradient descent and dropout

- $m \widehat{=}$ # iterations $\widehat{=}$ # data points randomly drawn with replacement $\widehat{=}$ # "thinned" subnets
- $K \widehat{=}$ # parameters $\theta_j$ to be estimated
- $\gamma \widehat{=}$ learning rate of the algorithm
- $J(\theta, \mu)_i \widehat{=}$ objective function for "thinned"sub-net $i$ depending on $\mu$

**Algorithm 2** stochastic gradient descent update dropout

randomly draw $m$ train data points from the train sample with replacement
**repeat**
  **for** $i = 1, ..., m$ **do**
    randomly draw mask $\mu$ for sub-net $i$
    **for** $j = 1, .., K$ **do**
      $\theta_j^{(i+1)} = \theta_j^{(i)} - \gamma \frac{\partial J(\theta, \mu)_i}{\partial \theta_j}$
    **end for**
  **end for**
**until** stop criterion is reached

# Test phase / deployment of dropout net

- Models output **probability distributions**: $p(y = y_j|x, \mu)$
- Bagging: arithmetic mean: $\tilde{p}_{ensemble}(y = y_j|x) = \frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y = y_j|x)$
- Dropout: more robust weighting via geometric mean:

$$\tilde{p}_{ensemble}(y = y_j|x) = \sqrt[2^d]{\prod_{\mu} p(y = y_j|x, \mu)} \text{ with } p(y = y_j|x, \mu) > 0$$

re-normalize for prediction:

$$p_{ensemble}(y = y_j|x) = \frac{\tilde{p}_{ensemble}(y = y_j|x)}{\sum_j \tilde{p}_{ensemble}(y = y_j|x)} \tag{16}$$

- Approximation of geometric mean by [Hinton et al., 2012]: **weight scaling rule**

# Test Dropout Net - Weight Scaling Rule

Approximate $p_{ensemble}$ by inspection of the **complete model**:
$\rightarrow$ multiply shared weights of the trained model coming out of unit $i$ by $p$

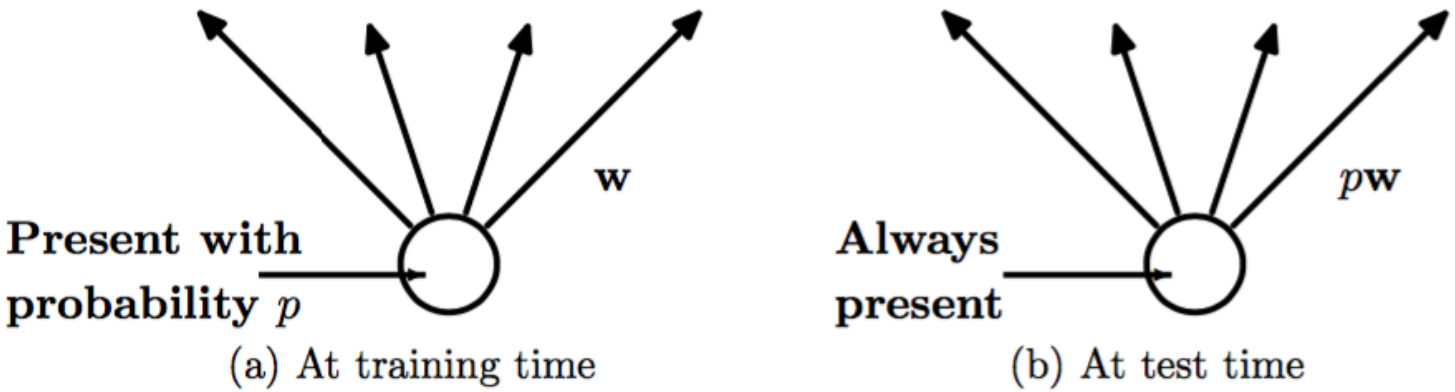



**Figure:** [Goodfellow et al., 2016]

[Goodfellow et al., 2016] show this can be proven to be **exact** for many classes of nets (e.g. Vanilla network with Softmax output layer)

# Bagging vs. Dropout

| | Bagging | Dropout |
|---|---|---|
| Train data | train data points drawn with replacement | |
| Basic Idea | model averaging | |
| # models | $k$ | possibly $2^d$ |
| Prediction | atrithmetic mean | geometric mean |
| Parameters | k independent models | parameter sharing |
| Train method | each model to convergence | each sub-net trained on single randomly sampled train data point restricted by $\mu$ |

# Notes on hyperparameters using dropout

Heuristics by [Srivastava et al., 2014]:

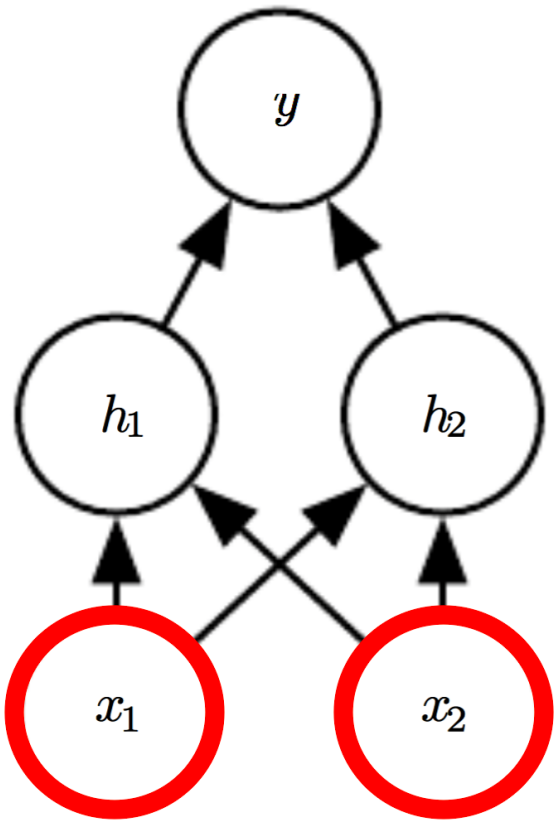
| Parameter | Description | Rule of thumb |
|---|---|---|
| Network size | Dropout rate reduces capacity of net/ number of hidden units | n/p |
| Learning rate | Dropout introduces lot of noise in the model: gradients cancel each other | 10-100 $\times$ standard $\gamma$ |
| Dropout rate | Probability of retaining a unit and $p = \pi$ in $\mu_q \sim Bernoulli(\pi)$ | $p = 0.8$ for inputs $p = 0.5$ for hidden units |
| Regularization | Combination with other regularization methods. **Max-norm-regularization** performs especially well | Max-norm-regularization |

R implementation: `h2o` and `deepnet`

# Structure

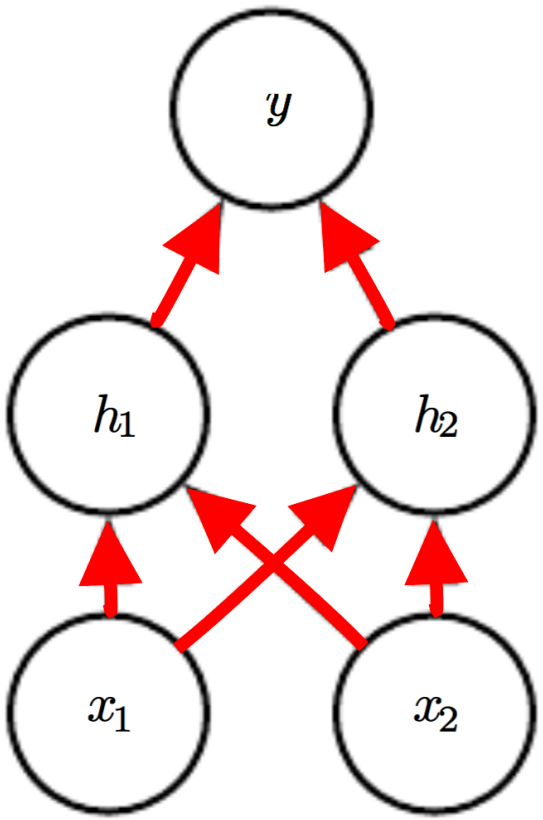
# Variations of Noise Injection

1. Inputs



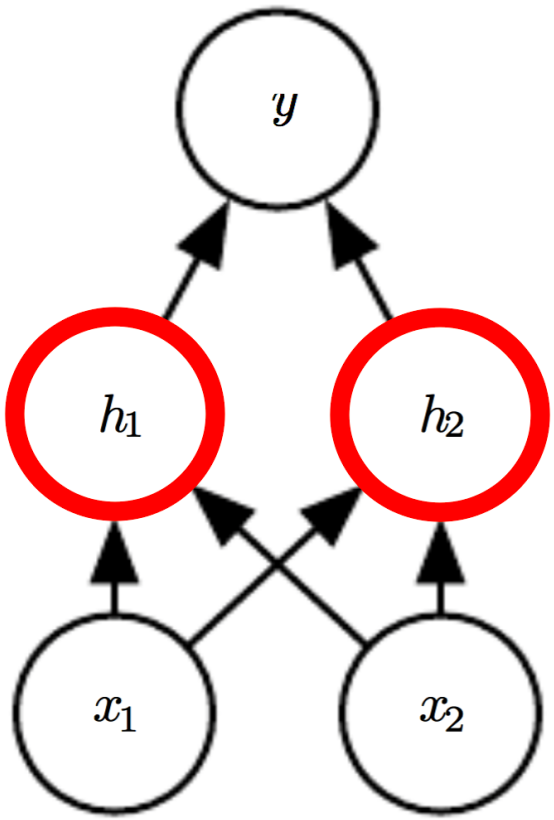
# Variations of Noise Injection

1. Inputs
2. Weights



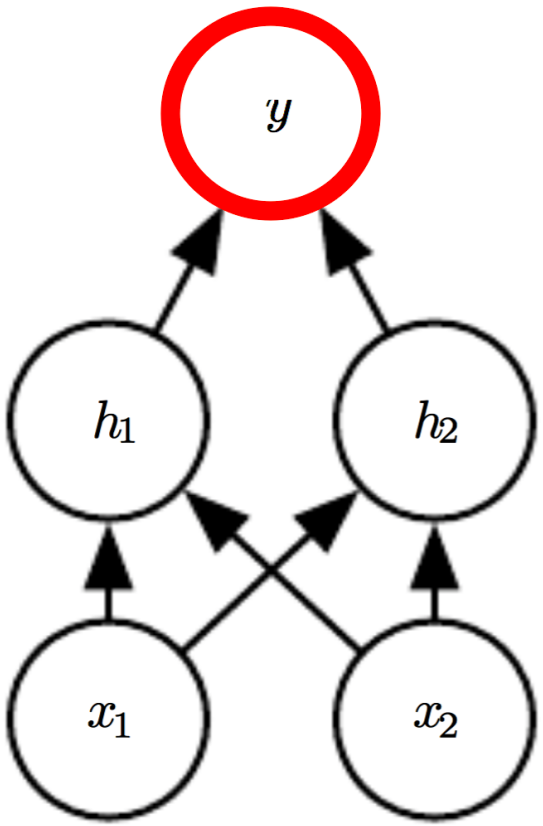
# Variations of Noise Injection

1. Inputs
2. Weights
3. Hidden units



# Variations of Noise Injection

1. Inputs
2. Weights
3. Hidden units
4. Outputs



# Noisy inputs

- Part of the data set augmentation strategy to amplify train data
- [Bishop, 1995] proofs: addition of noise on inputs with small variance is equivalent to norm penalty on the weights

# Noisy weights

- Regression setting, minimize MSE: $J = \mathbb{E}_{p(x,y)}[(\hat{y}(x) - y)^2]$
- Perturbation on weights such that $\tilde{\omega} = \omega + \epsilon_\omega$ with $\epsilon_\omega \sim \mathcal{N}(0, \eta\mathcal{I})$
- [Hochreiter et al., 1995] proof:

$$\nabla_\omega \tilde{J}_\omega = \nabla_\omega J + \eta \mathbb{E}_{p(x,y)} \left[ H^2 \right]$$

- Intuition:
  - Penalization of high levels of the Hessian
  - Weights pushed to areas with "flatter" minima
  - "Perfect" minima cannot be found because of noise, but approximations lie in a "flat" region
  - Weights more insensitive to variation as neighbor weights similarly low

# Noisy hidden units

- Dropout-mask $\mu$ can be interpreted as noise added to the hidden units in each SGD step
- Units cannot rely on each other and therefore have to diversify
- Not just simple addition of white noise to inputs
- Adaptive and intelligent destruction of information
- Optimisation strategy Batch Normalization applies noise to improve optimization which has regularizing side effect
- Illustrational examples
  - Focus on nose in face recogniton
  - Metaphor with eyes: blind spots vs. blurred vision

# Noisy outputs

- Idea: incorporate mesaure errors on the output labels
- Most known method: **label smoothing**
- Assume model with softmax output function and $k$ labels
- Originally: hard targets $y_j = \{0, 1\}$
- Relax them to softer targets $y_j = \{\frac{\epsilon}{k-1}, 1-\epsilon\}$
- Widely known and applied strategy

# Structure

# Basic concept

- Artificially construct data that only differ little from originals but confuse the net
- Use this **adversarial training data** to further train the model and enhance its performance

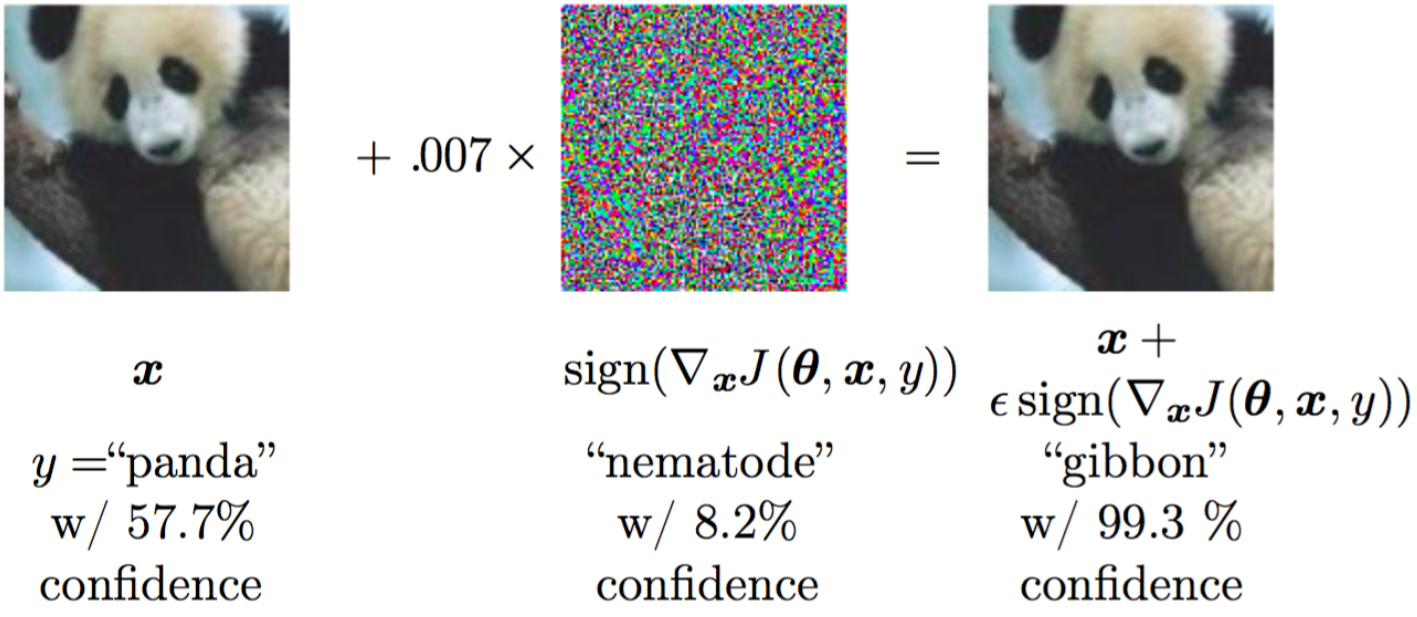



**Figure:** [Goodfellow et al., 2016]

# Structure

# About the data

- MNIST data set by Yann le Cun
- Easy access using package `darch`
- 70K images of handwritten digits in $28 \times 28$ pixels format
- Split in 60K train and 10K test data
- Classification task with 10 classes
- One of most popular benchmark data sets for image recognition

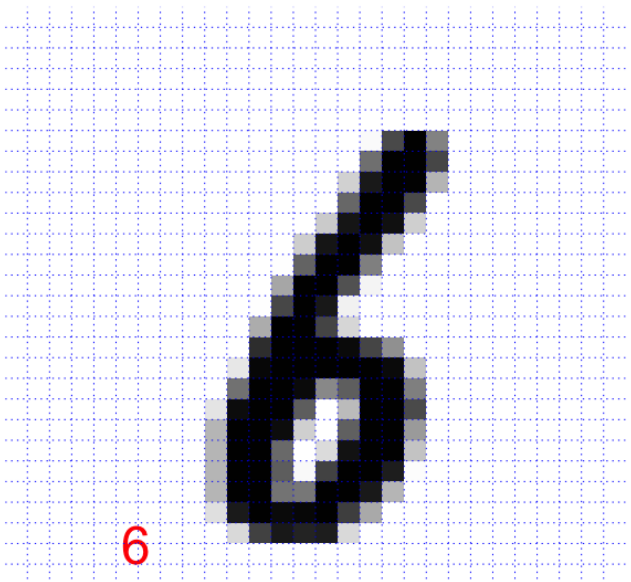



**Figure:** Example image

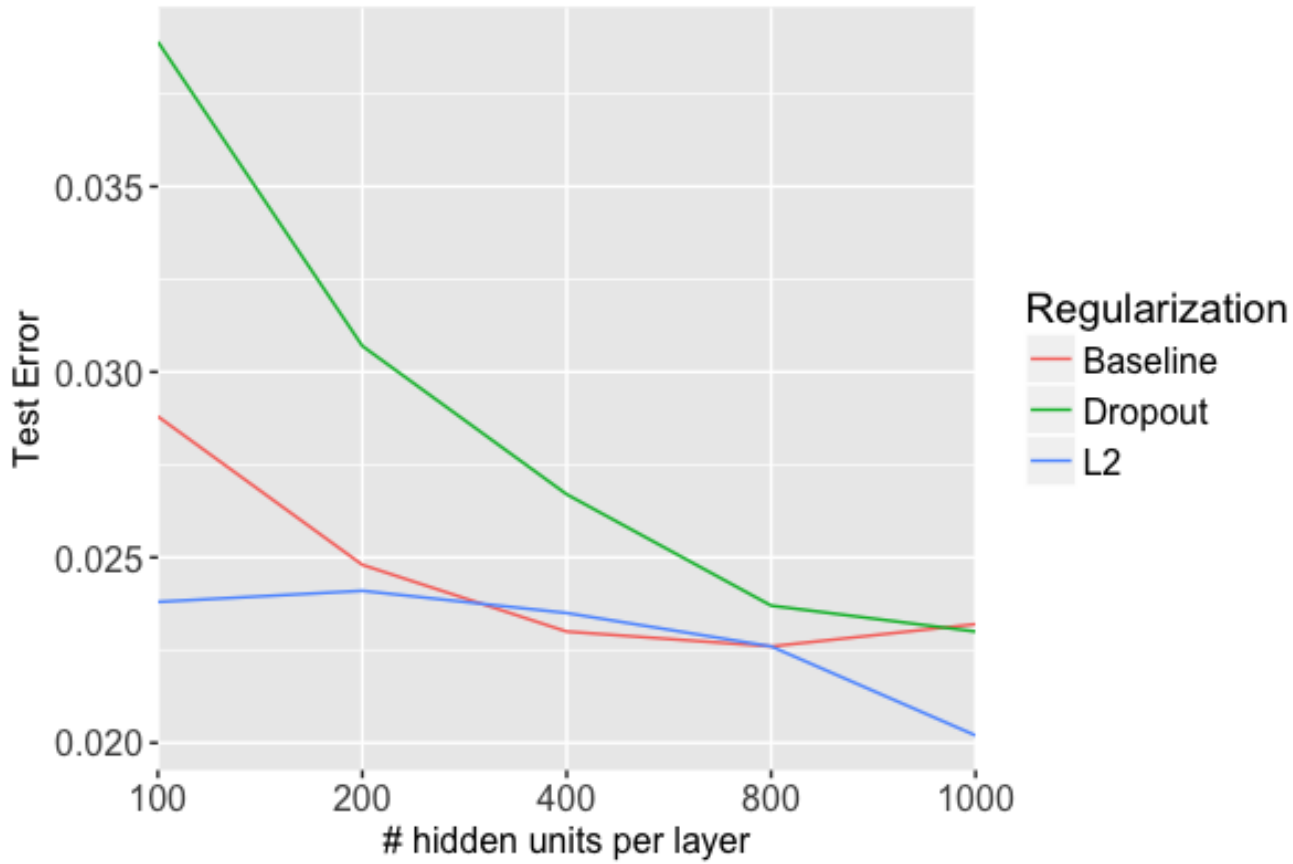
# About the model

- Neural net created using function `deeplearning` from package `h2o` called via `mlr`
- Feed-forward net with specifications:

| | |
|---|---|
| Activation function | Rectifier |
| Epochs | 10 |
| # hidden layer | 2 |
| Loss funtion | Cross entropy |
| Learning rate | 0.001 |
| Error measure | Mean misclassification error |

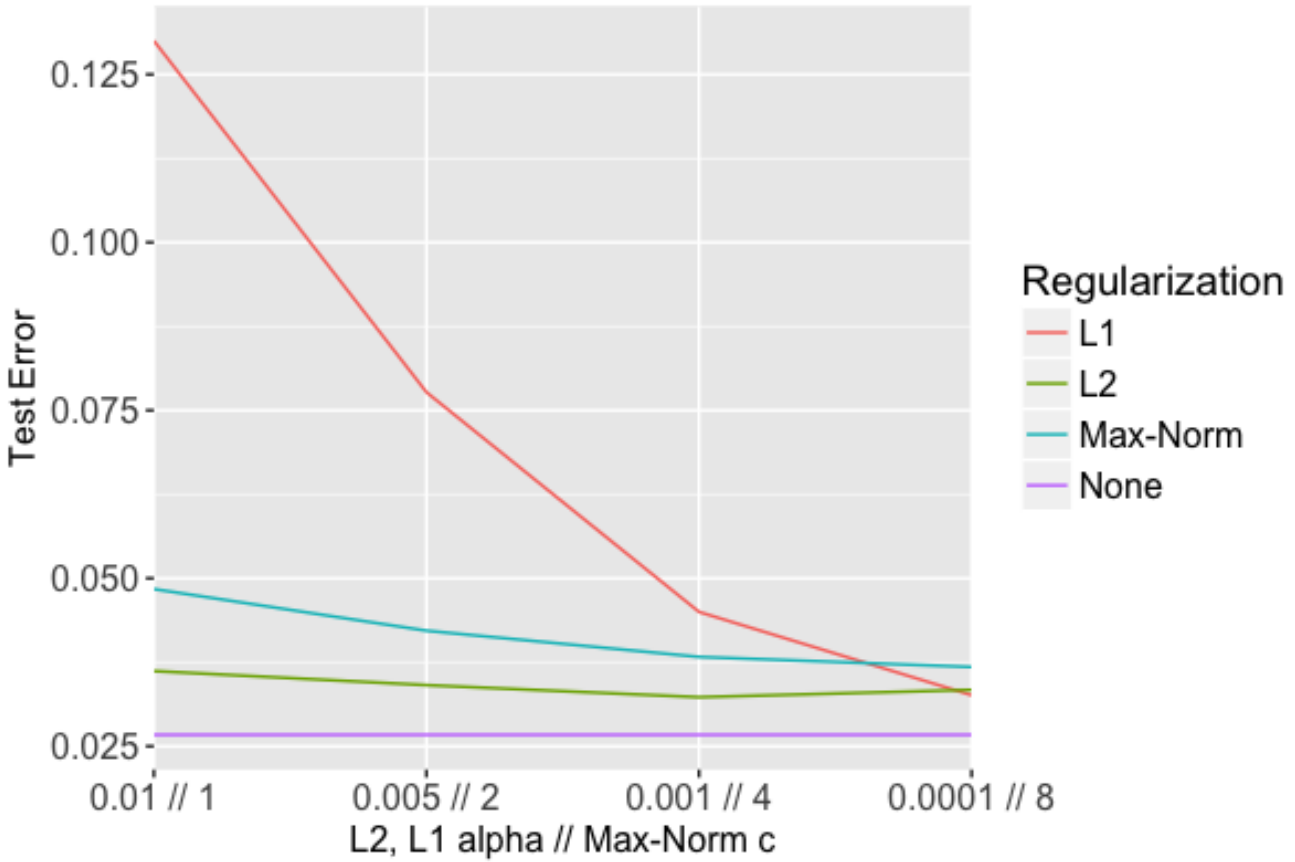
# Structure

# Performance with varying net-size




- 2-layer neural net with varying # hidden units
- Dropout with $p_{\text{hidden\_units}} = 0.5$ and $p_{\text{input\_units}} = 0.8$
- $L^2$-Regularization with penalization rate $\alpha = .001$

# Dropout combined with further regularization




- 2-layer net with 400 hidden units each
- $\Rightarrow$ Max-Norm seems **not to** work best combined with Dropout

# Dataset augmentation

- Train model on three data sets:
  - Small stratified subset with 30K images
  - Original train set with 60K images
  - Vertically augmented data set with 180K images
- Evaluate on same test sample with 10K images

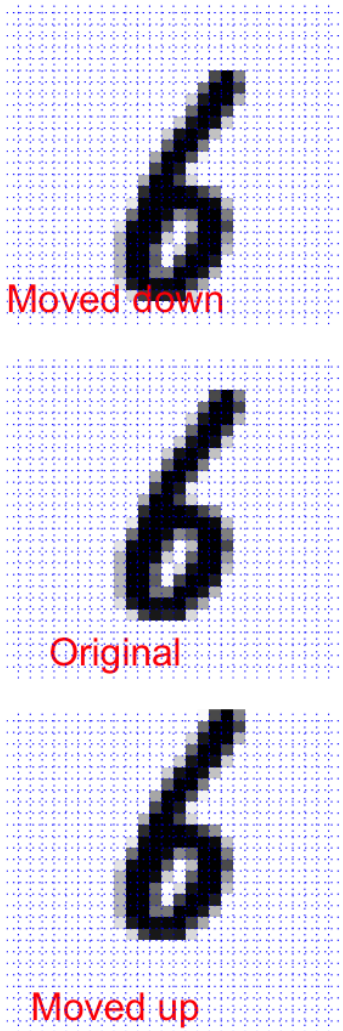



**Figure:** Augmented digit

# Dataset augmentation

- Train model on three data sets:
  - Small stratified subset with 30K images
  - Original train set with 60K images
  - Vertically augmented data set with 180K images
- Evaluate on same test sample with 10K images

| | 30K | 60K | 180K |
|---|---|---|---|
| Baseline | 3.26 % | 2.53 % | 2.20 % |
| Dropout | 4.80 % | 3.67 % | 3.60% |
| L2 | 3.23 % | 2.39 % | 2.05 % |

# Structure

# Current research on regularization

- Dense-sparse-dense neural nets by [Han et al., 2016]
- Shakeout by [Kang et al., 2016]
- Eigenvalue decay by [Ludwig et al., 2014]
- Drop-connect by [Wan et al., 2013]

# Thank you


Bishop, C. M. (1995).
Training with noise is equivalent to tikhonov regularization.
*Neural computation*, 7(1):108–116.


Cui, X., Goel, V., and Kingsbury, B. (2015).
Data augmentation for deep neural network acoustic modeling.
*Transactions on Audio, Speech and Language Processing (TASLP)*, 23(9):1469–1477.


Goodfellow, I., Bengio, Y., and Courville, A. (2016).
Deep learning.
Book in preparation for MIT Press.


Han, S., Pool, J., Narang, S., Mao, H., Tang, S., Elsen, E., Catanzaro, B., Tran, J., and Dally, W. J. (2016).
Dsd: Regularizing deep neural networks with dense-sparse-dense training flow.
*arXiv preprint arXiv:1607.04381*.


Hastie, T., Tibshirani, R., and Friedman, J. H. (2009).
*The elements of statistical learning*.
Springer series in statistics. Springer, New York [u.a.].


Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012).
Improving neural networks by preventing co-adaptation of feature detectors.
*CoRR*, abs/1207.0580.


Hochreiter, S., Schmidhuber, J., et al. (1995).
Simplifying neural nets by discovering flat minima.
*Advances in Neural Information Processing Systems*, pages 529–536.


Kang, G., Li, J., and Tao, D. (2016).
Shakeout: A new regularized deep neural network training scheme.
In *Thirtieth AAAI Conference on Artificial Intelligence*.

 Ludwig, O., Nunes, U., and Araujo, R. (2014).
Eigenvalue decay: A new method for neural network regularization.
*Neurocomputing*, 124:33 – 42.

 Shen, J., Xu, H., and Li, P. (2014).
Online optimization for large-scale max-norm regularization.
*ArXiv e-prints*.

 Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014).
Dropout: A simple way to prevent neural networks from overfitting.
*Journal of Machine Learning Research*, 15:1929–1958.

 Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013).
Regularization of neural networks using dropconnect.
In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066.


Wu, R., Yan, S., and Shan, Y. (2015).
Deep image: Scaling up image recognition.
*CoRR*, abs/1501.02876.

# Appendix 1: weight decay as early stopping I

- Quadratically approximate $J(\theta)$ at minimal optimal value $\omega^*$ and $\theta = \omega$:

$$\hat{J}(\theta) = J(\omega^*) + \frac{1}{2}(\omega - \omega^*)H(\omega - \omega^*)^T$$

with $H$ being the Hessian of $J$ at $\omega^*$ leading to gradient:

$$\nabla_\omega \hat{J}(\omega) = H(\omega - \omega^*)$$

- Examine the gradient descent update on $\hat{J}$ assuming $\omega^{(0)} = 0$ :

$$\omega^{(m)} \leftarrow \omega^{(m-1)} - \gamma \nabla_\omega \hat{J}(\omega^{(m-1)})$$
$$\omega^{(m)} \leftarrow \omega^{(m-1)} - \gamma H(\omega^{(m-1)} - \omega^*) \mid -\omega^*$$
$$\omega^{(m)} - \omega^* \leftarrow (I - \gamma H)(\omega^{(m-1)} - \omega^*)$$

# Appendix 1: weight decay as early stopping II

- Eigendecompose $H = Q\Lambda Q^T$ such that:

$$\omega^{(m)} - \omega^* \leftarrow (I - \gamma Q\Lambda Q^T)(\omega^{(m-1)} - \omega^*)$$
$$Q^T(\omega^{(m)} - \omega^*) \leftarrow (I - \gamma\Lambda)Q^T(\omega^{(m-1)} - \omega^*)$$

- Assuming $\omega^0 = 0$ and $\gamma$ such that $|1 - \gamma\lambda_i| < 1$ leads to:

$$Q^T\omega^{(m)} = [I - (I - \gamma\Lambda)^{(m)}]Q^T\omega^*$$

- Recap the formula from the $L^2$ penalization:

$$\tilde{\omega} = Q(\Lambda + \alpha I)^{-1}\Lambda Q^T\omega^*$$

and rearrange it to:

$$Q^T\tilde{\omega} = [I - (\Lambda + \alpha I)^{-1}\alpha]Q^T\omega^*$$

- $\Rightarrow$ weight decay = early stopping if $\gamma, m, \alpha$ chosen such that:

$$(I - \gamma\Lambda)^{(m)} = (\Lambda + \alpha I)^{-1}\alpha$$

# Appendix 1: weight decay as early stopping III

- Examine the effect for one weight $\tilde{\omega}_i$:

$$(1-\gamma\lambda_i)^{(m)} = (\lambda_i+\alpha)^{-1}\alpha|log()$$

$$mlog(1+(-\gamma\lambda_i)) = log(\frac{\alpha}{\lambda_i+\alpha}) = -log(1+\frac{\lambda_i}{\alpha})$$

use series expansion for $log(1+x)$ assuming $|\gamma\lambda_i| \ll 1$ and $|\frac{\lambda_i}{\alpha}| \ll 1$ :

$$\Rightarrow m \approx \frac{1}{\gamma\alpha} \Leftrightarrow \alpha \approx \frac{1}{m\gamma}$$

- Recap weight decay: weights corresponding to directions of high curvature (high $\lambda_i$, required correction) are less regularized by weight decay than directions of low curvature (low $\lambda_i$, "unnecessary" update)
- Weights from the first case learn early relative to the latter

# Appendix 1: weight decay as early stopping IV

- Graphic interpretation:

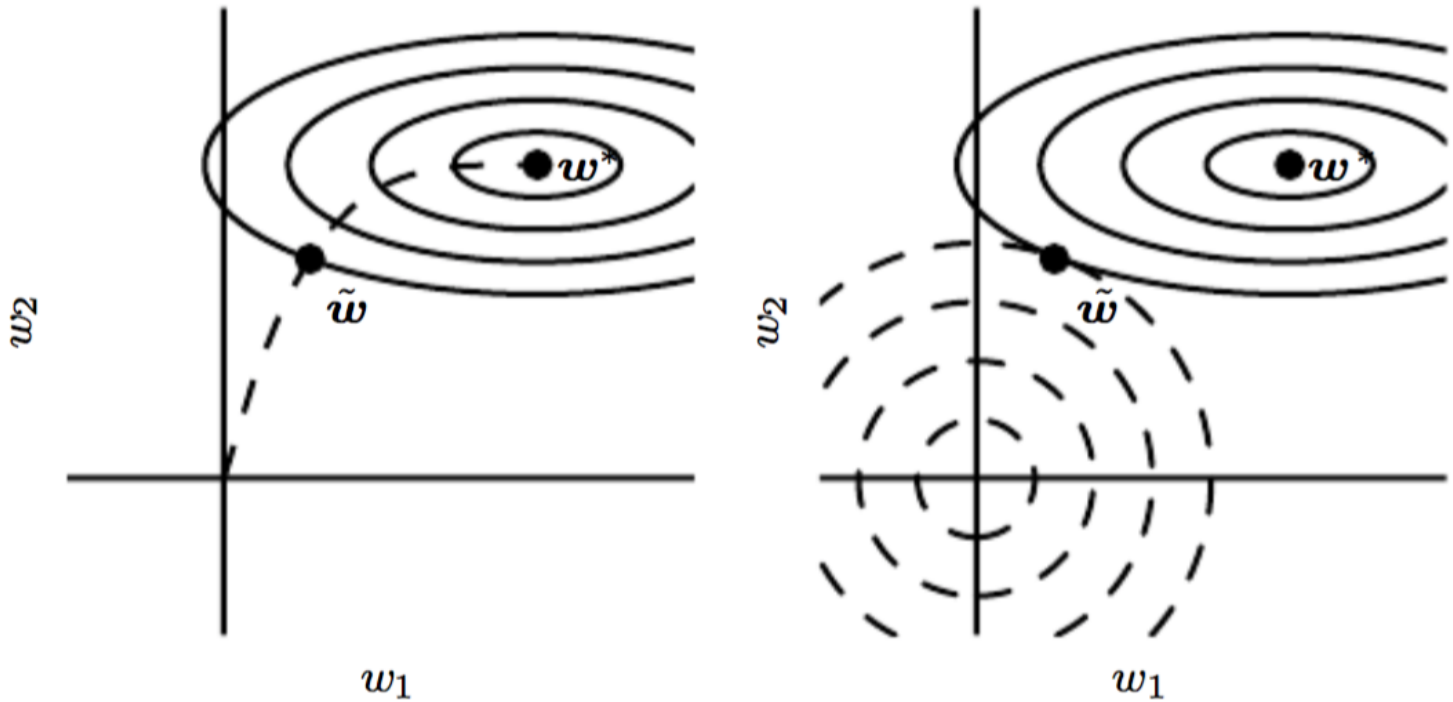



**Figure:** [Goodfellow et al., 2016]

# Appendix 2: formal algorithm early stopping

Set $\theta_0 \widehat{=}$ initial parameters, $n \widehat{=}$ step size, $p \widehat{=}$ patience, $v \widehat{=}$ validation set error

**Algorithm 3** Early stopping meta algorithm

Initialize $\theta^* \leftarrow \theta \leftarrow \theta_0$ and $m, j \leftarrow 0$ and $v \leftarrow \infty$ and $m^* \leftarrow m$
**while** $j < p$ **do**
    run update algorithm for $\theta$ n times
    $m \leftarrow m + n$ and $v' \leftarrow J(\theta, data^{validation})$
    **if** $v' < v$ **then**
        $j \leftarrow 0$ and $\theta^* \leftarrow \theta$ and $m^* \leftarrow m$ and $v \leftarrow v'$
    **else**
        $j \leftarrow j + 1$
    **end if**
**end while**
optimal parameters are $\theta^*$ and optimal number of iterations is $m^*$

# Appendix 3: illustration geometric mean in dropout

| | $P(y = y_1\|x)$ | $P(y = y_2\|x)$ | $P(y = y_3\|x)$ | $\sum$ |
|---|---|---|---|---|
| Model 1 | 0.20 | 0.70 | 0.10 | 1.00 |
| Model 2 | 0.10 | 0.80 | 0.10 | 1.00 |
| Model 3 | 0.05 | 0.90 | 0.05 | 1.00 |
| Model 4 | 0.05 | 0.90 | 0.05 | 1.00 |
| **Model 5** | **0.80** | **0.10** | **0.10** | **1.00** |
| Arithmetic mean | 0.24 | 0.68 | 0.08 | 1.00 |
| Geometric mean | 0.13 | 0.54 | 0.08 | 0.75 |
| Re-normalized | 0.18 | 0.72 | 0.10 | 1.00 |

$$mean_{arithmetic}(x_1, ..., x_n) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$mean_{geometric}(x_1, ..., x_n) = \sqrt[n]{\prod_{i=1}^{n} x_i} \text{ with } x_i > 0 \forall i = 1, .., n$$