MSE | MASTER OF SCIENCE IN ENGINEERING

Regularisation and Faster Optimisers

TSM_DeLearn

Jean Hennebert
Martin Melchior

Overview

Recap from last week

Vanishing and Exploding Gradients

Regularisation to Avoid Overfitting

Faster Optimisation Schemes

Recap from last week

backprop

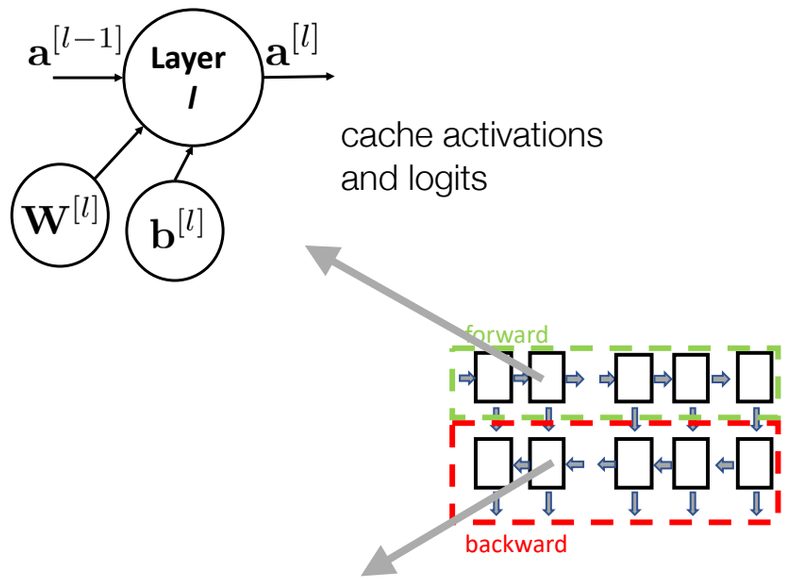
Important Points to Resume from Last Weeks

- Week 4:
  - Learned about model selection and the tools to detect overfitting: Searching in hyper-parameter space is computationally costly!
  - Curse of dimensionality and the benefits of using deep learning: Hierarchy of concepts

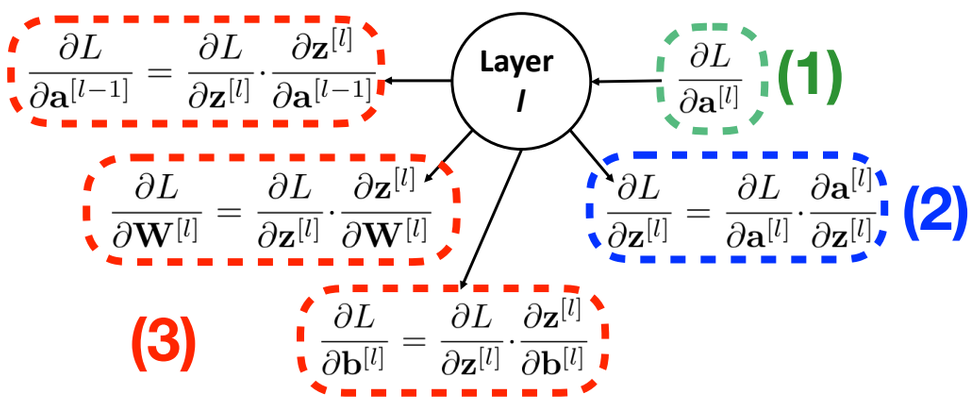
- Week 5:
  - Backprop as an efficient scheme for computing gradients in deep models.

Forward and Backprop

Forward Propagation
Activations are computed from the previous layer activations.



Back Propagation
As a result of the chain rule of calculus, gradients of the loss w.r.t. activations of the previous layer can be computed from the gradient w.r.t. the activations of the given layer. Furthermore, gradients w.r.t. weights, biases can be computed from the gradient w.r.t. the activations of the given layer. Hence for computing all gradients, we can propagate the gradient w.r.t. the activations back through all the layers and compute the gradients w.r.t. weights and biases on the fly.



Useful Resources

Andrew Ng, Deep Learning, Coursera
https://www.coursera.org/learn/neural-networks-deep-learning
(you don't need to pay for just watching)
Specifically for computational graphs and backprop:

- Derivatives with computational graph: https://www.coursera.org/learn/neural-networks-deep-learning/lecture/0VSHe/derivatives-with-a-computation-graph
- Backprop intuition: https://www.coursera.org/learn/neural-networks-deep-learning/lecture/6dDj7/backpropagation-intuition-optional
- Forwardprop: https://www.coursera.org/learn/neural-networks-deep-learning/lecture/MijzH/forward-propagation-in-a-deep-network
- Getting the dimensions of the arrays correct: https://www.coursera.org/learn/neural-networks-deep-learning/lecture/Rz47X/getting-your-matrix-dimensions-right
- Forwardprop/backprop: https://www.coursera.org/learn/neural-networks-deep-learning/lecture/znwiG/forward-and-backward-propagation

Fei-Fei Li, Convolutional Neural Networks, Stanford University
course's web site: http://cs231n.stanford.edu/
lecture videos (2017): https://www.youtube.com/playlist?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv

- Computational Graph and Backdrop: https://www.youtube.com/watch?v=d14TUNcbn1k&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=4

Google Developers Crash Course
https://google-developers.appspot.com/machine-learning/crash-course/

- Nice Animation of Forward Prop and Backprop: https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/

Plan for this week

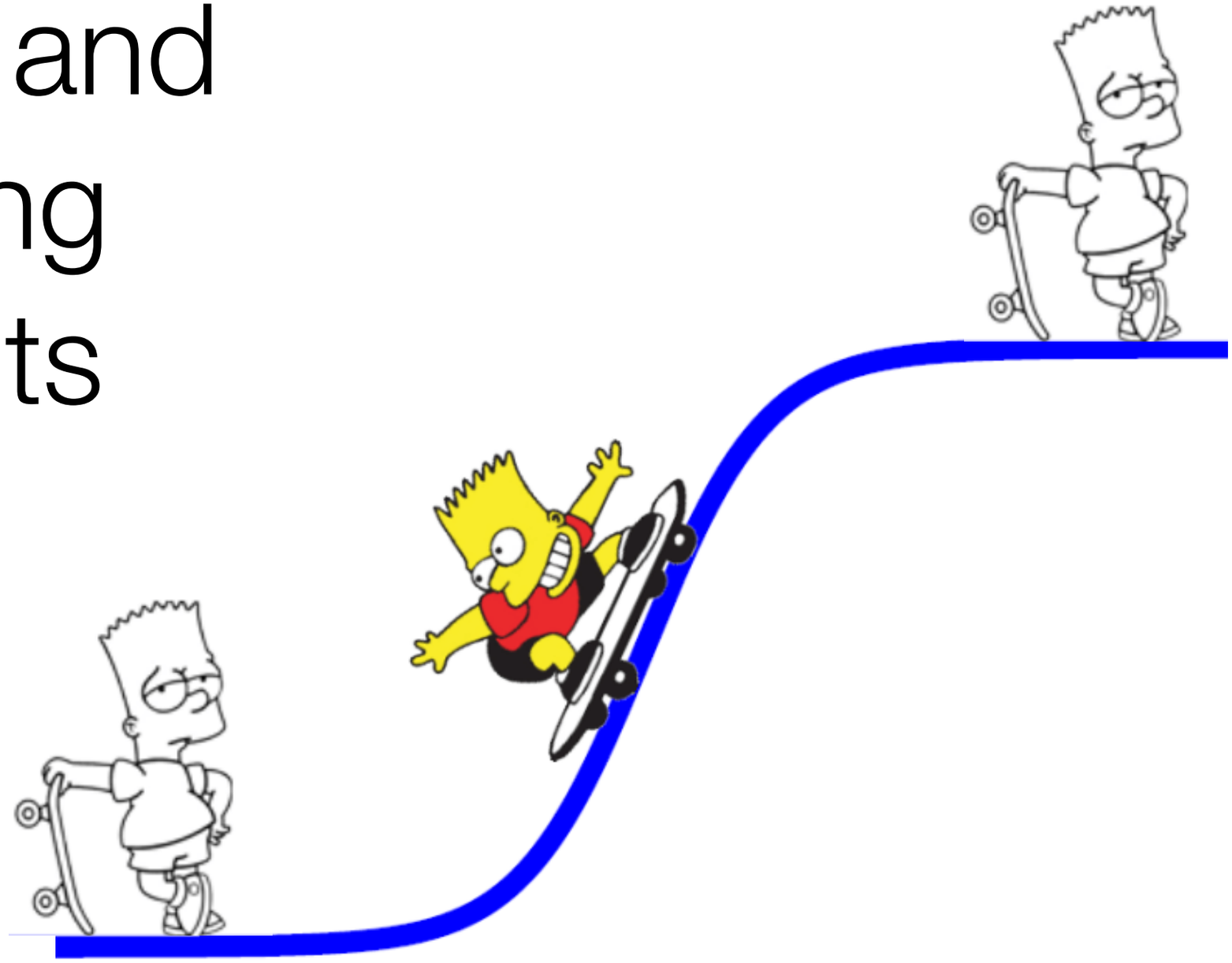
With the back-propagation scheme, we have an efficient algorithm used for training deep neural nets. Unfortunately, further issues are encountered - to be addressed in this and next week.

- Understand the **vanishing/exploding gradients** issue encountered with deep models and how it can be opposed.

- Get to know **regularisation** methods as a means to counter the problem of overfitting.

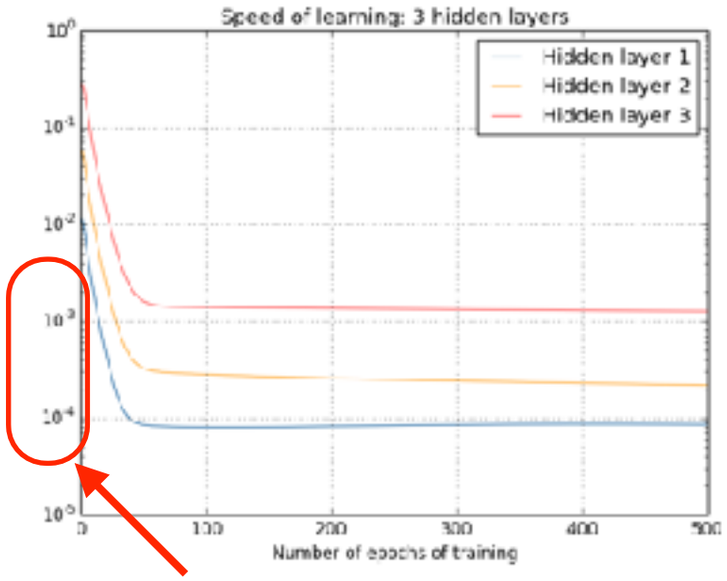
- Learn about **improved optimisation schemes** and what issues these try to overcome.

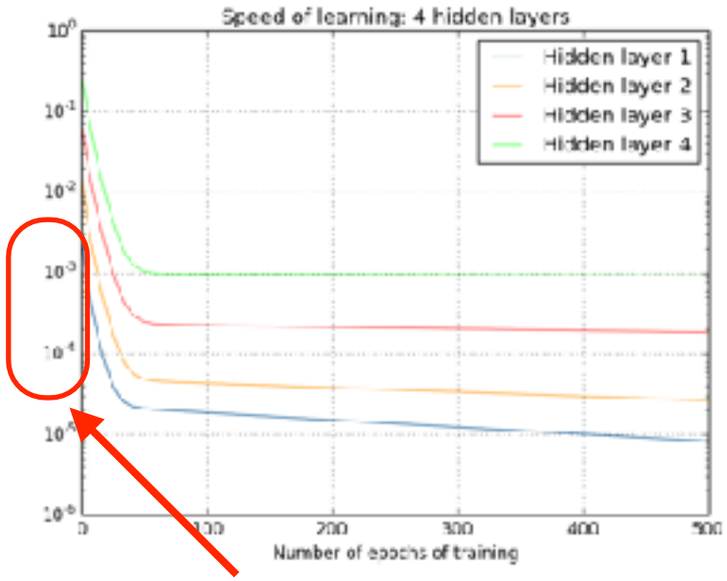
Vanishing and Exploding Gradients



Small Learning Speed at Early Layers

- When learning MNIST with backprop and standard normally distributed initial weights, we observe that (see http://neuralnetworksanddeeplearning.com/chap5.html)
  - the learning is slower in earlier layers;
  - the learning becomes even slower when having more layers.





Small gradients here don't imply that we are close to the minimum!

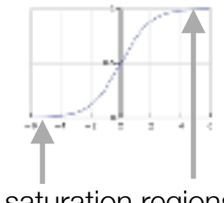
Learning speed (~length of gradients) in the first two layers drops by an order of magnitude when using four instead of three hidden layers (see the different scales in the two figures)!

- Sometimes, the opposite i.e. exploding gradients are observed in earlier layers of deep neural nets. This happens mostly in recurrent deep nets.
- Generally, the gradients in deep neural networks are *unstable*, tending to either vanish or explode in earlier layers (vanishing is prevalent).

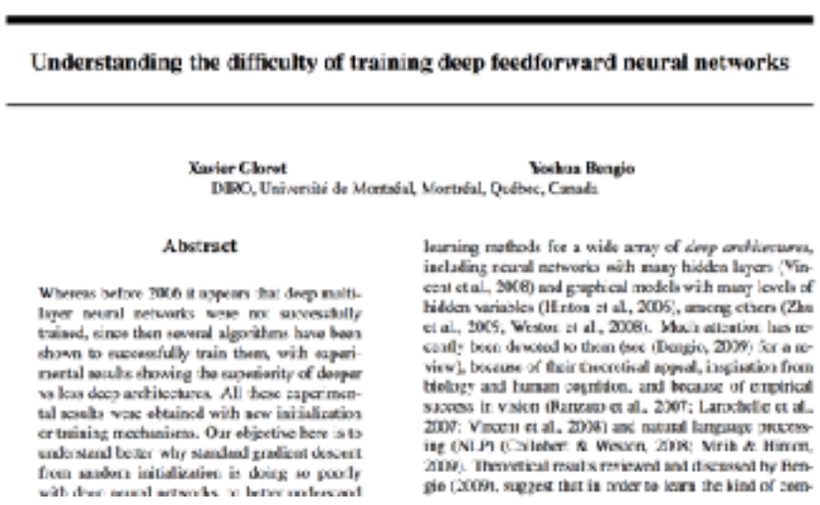
Unstable Gradients

- This behaviour makes the learning of deep network very difficult. It was one of the reasons why deep neural networks were mostly abandoned for some time.

- Paper(1) by Glorot and Bengio in 2010 gives some insight into what happens:
  - With the settings typically used in 2010:
    - *sigmoid* activation function
    - weights initialised with standard normally distributed random numbers ($\mu = 0, \sigma = 1$)
  - The variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers.
  - It is very likely to end up in the higher layers in the saturation regions of the activation function during the forward propagation - where the gradients are zero.



saturation regions



Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot
Yoshua Bengio
DIRO, Université de Montréal, Montréal, Québec, Canada

Abstract

Whereas before 2006 it appears that deep multi-layer neural networks were not successfully trained, since then several algorithms have been shown to successfully train them, with experimental results showing the superiority of deeper vs less deep architectures. All these experimental results were obtained with new initialization or training mechanisms. Our objective here is to understand better why standard gradient descent from random initialization is doing so poorly with deep neural networks, to better understand

learning methods for a wide array of *deep architectures*, including neural networks with many hidden layers (Vincent et al., 2008) and graphical models with many levels of hidden variables (Hinton et al., 2006), among others (Zhu et al., 2005; Weston et al., 2008). Much attention has recently been devoted to them (see (Bengio, 2009) for a review), because of their theoretical appeal, inspiration from biology and human cognition, and because of empirical success in vision (Ranzato et al., 2007; Larochelle et al., 2007; Vincent et al., 2008) and natural language processing (NLP) (Collobert & Weston, 2008; Mnih & Hinton, 2009). Theoretical results reviewed and discussed by Bengio (2009), suggest that in order to learn the kind of com-

The authors claim that

- the variance of the outputs of each layer should be equal to the variance of its inputs,
- the gradients should have equal variance before and after flowing through a layer in the reverse direction.

(1) http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Intuition behind Vanishing Gradients

Source: http://neuralnetworksanddeeplearning.com/chap5.html

- **Multiplicative Structure** of the expression for the gradients w.r.t. weights and biases (as seen last week this is a consequence of the chain rule).
  - Toy MLP example with n identical layers each with a single neuron and sigmoid activation function:

$$\frac{\partial L}{\partial w^{[l]}} = a^{[l-1]} \cdot g'(z^{[l]}) \cdot \left(\prod_{k=l+1}^{L} w^{[k]} g'(z^{[k]})\right) \cdot \frac{\partial L}{\partial a^{[L]}}$$

$$\frac{\partial L}{\partial b^{[l]}} = g'(z^{[l]}) \cdot \left(\prod_{k=l+1}^{L} w^{[k]} g'(z^{[k]})\right) \cdot \frac{\partial L}{\partial a^{[L]}}$$

Product Term

- **Product Term** is responsible for the gradients to shrink to zero or to become large in deep nets during the learning process.
  For sigmoid activation functions: $|g'(z)| \leq g'(0) = 1/4$. With random weights initialisation (standard normal) the weights $w \in [-2, 2]$ with 95% probability so that, with high probability, each of the factors in the product term is small and, for early layers in a deep net, the product is very small.

- **Coordinating Updates**: The product term implies that the gradients in early layers contain a factor in common with the later layers. This introduces significant instability — updates of the weights in different layers are highly coupled.

Towards Fixing the Problem

The problem cannot be completely solved — but alleviated…

Methods:

- **Parameter Initialisation**: Properly initialise the weights so that the logits z do not grow too large in magnitude.
- **Batch Normalisation, Gradient Clipping**: Make sure that the weights do not grow too large in magnitude also during training.
- **Suitable Activation Functions**: Use activation functions that cannot saturate.

Parameter Initialisation, Xavier/He Initialisation

- **Randomly initialise weights**: To break symmetries at start of learning.

- **Put initial weights at proper scale**: Make sure that the variance is kept stable across layers - ideally in both directions. Choose the weights such that the z-values

$$z^{[l]} = w_1^{[l]} a_1^{[l-1]} + \cdots + w_{n_{l-1}}^{[l]} a_{n_{l-1}}^{[l-1]}$$

in the different layers are normalised - by properly scaling the width of the distribution. Specific scaling depends on the random distribution used to generate the weights and the activation function:

	Activation function	Uniform distribution [−r, r]	Normal distribution
Xavier	Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Xavier	Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
He	ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

number of input and output connections **for a given layer**

"Hands-On Machine Learning with Scikit-Learn and TensorFlow", A. Géron (O'Reilly), 2017.

- Scheme does not guarantee to keep the input/output variance identical in both directions, but works well in practice.

Batch Normalisation

- Idea introduced by S. Ioffe and C. Szegedy(*):
  - Normalise the input to each layer per mini-batch by adding an operation in the model just before the activation function of each layer: zero-center and scale the inputs by estimating mean and stdev from the current mini-batch.
  - Then let the model learn the optimal scale and mean of the inputs for each layer.

- Effect:
  - Significantly reduces the problem of coordinating updates across many layers.
  - Limits the amount to which updating the parameters in the earlier layers can affect the distribution of values the later layers see.

- Can be applied to any input or hidden layer in a network.

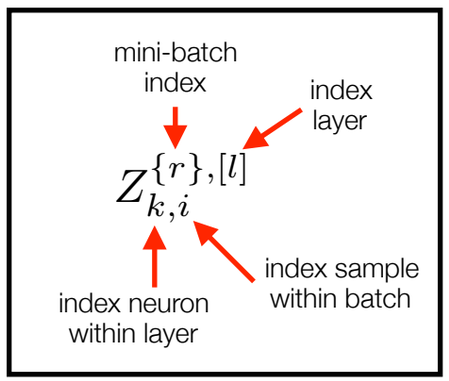
(*) “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” S. Ioffe and C. Szegedy (2015).

Batch Normalisation Equations (1)

Ioffe and Szegedi recommend to apply batch normalisation to the inputs to the activation function, i.e. the logits $z_k^{[l]}$.

Normalisation per mini-batch:

$$\left(Z_{\text{norm}}^{\{r\},[l]}\right)_{k,i} = \frac{Z_{k,i}^{\{r\},[l]} - \mu_k^{\{r\},[l]}}{\sigma_k^{\{r\},[l]} + \epsilon}$$



where $\mu_k^{\{r\},[l]}$ and $\sigma_k^{\{r\},[l]}$ are given by

$$\begin{aligned}\mu_k^{\{r\},[l]} &= \frac{1}{N_B}\sum_{i=1}^{N_B} Z_{k,i}^{\{r\},[l]} \\ \sigma_k^{\{r\},[l]} &= \frac{1}{N_B}\sum_{i=1}^{N_B}\left(Z_{k,i}^{\{r\},[l]} - \mu_k^{\{r\},[l]}\right)^2\end{aligned}$$

Scaling and **Shifting**:

$$\hat{Z}_{k,i}^{\{r\},[l]} = \gamma_k^{[l]}\left(Z_{\text{norm}}^{\{r\},[l]}\right)_{k,i} + \beta_k^{[l]}$$

where $\gamma_k^{[l]}$ and $\beta_k^{[l]}$ are parameters to be learned (i.e. optimised)

When using batch normalisation, the parameter $b_k^{[l]}$ used in the definition of $Z_{k,i}^{\{r\},[l]}$ is eliminated ($b_k^{[l]} = 0$) since it is redundant.

Batch Normalisation Equations (2)

- When using batch normalisation, the parameter $b_k^{[l]}$ used in the definition of $Z_{k,i}^{\{r\},[l]}$ is eliminated ($b_k^{[l]} = 0$) since it is redundant:

$$Z_{k,i}^{\{r\},[l]} = \sum_{j=1}^{n_{l-1}} W_{k,j}^{[l]} \cdot A_{j,i}^{\{r\},[l-1]}$$

- Batch Normalisation at Test Time or in Production
  - The mean $\mu_k^{\{r\},[l]}$ and stdev $\sigma_k^{\{r\},[l]}$ computed per mini-batch during training are used to compute suitable averages and stdev's that can be used at test time or for production. Typically, an exponentially weighted average is used — for a parameter ρ the update rule is defined through a running average $\hat{\rho}^{(r)}$ and decay rate ν by

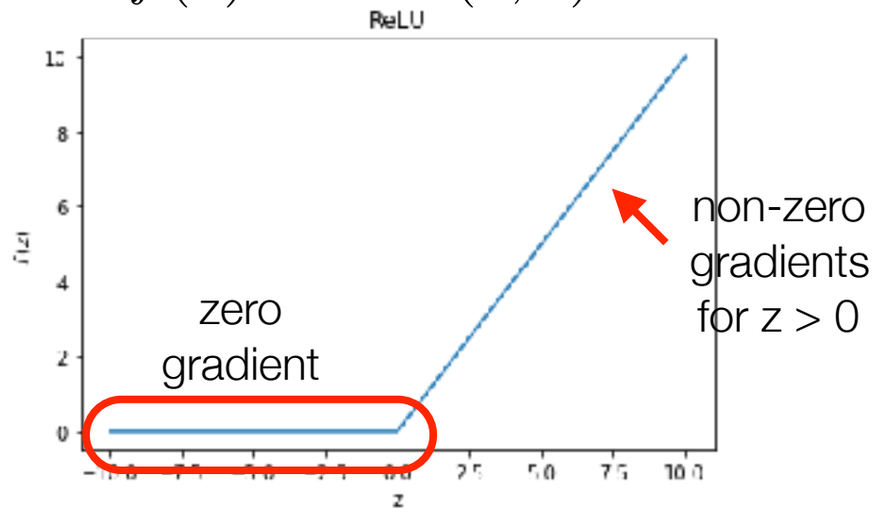
$$\hat{\rho}^{(r+1)} = (1 - \nu)\hat{\rho}^{(r)} + \nu\rho$$

- Batch normalisation changes backprop equations
  - The derivates w.r.t. to the new variables $\gamma_k^{[l]}$, $\beta_k^{[l]}$ need to be included.
  - The derivatives of the estimates $\mu_k^{\{r\},[l]}$ and $\sigma_k^{\{r\},[l]}$ also need to be taken into account.

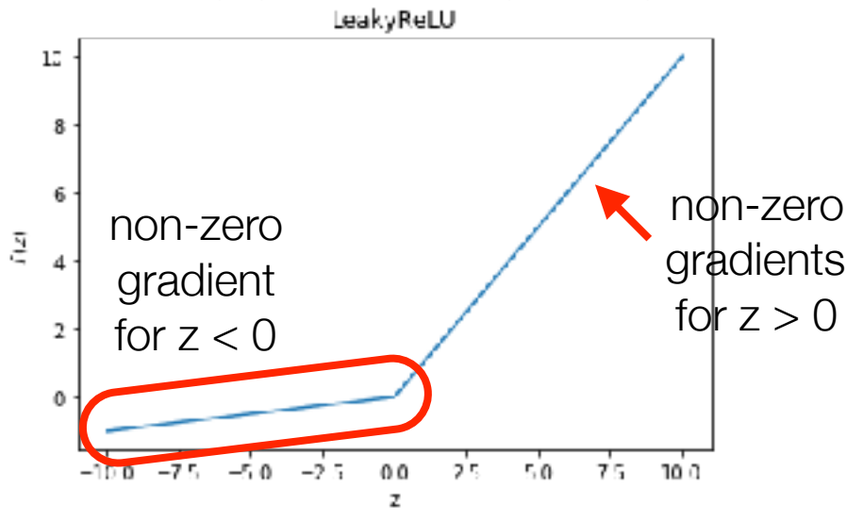
Non-Saturating Activation Functions

Avoid using S-shaped activation functions that flatten out in regions with larger magnitudes of z. This can be achieved by ReLU, LeakyReLU or ELU (see Week 3).

$$f(z) = \max(0, z)$$



$$f(z) = \max(\eta z, z)$$



ReLU suffers the ***dying units problem***: During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it is outputting 0. Since the gradient at z<0 is 0, there is no weight update for this neuron and the neuron is likely to stay dead.

With the **leaky ReLU** (or its smooth version, the ELU) this problem cannot occur. The parameter η can be determined by hyper-parameter tuning or by setting a good default value ($\eta = 0.01$). See (*) for an evaluation of the different ReLUs.

(*) "Empirical Evaluation of Rectified Activations in Convolution Network," B. Xu et al. (2015).

Gradient Clipping

- A simple technique to counter exploding gradients is to simply clip the gradients in length during backprop so that they never exceed some threshold.

- Neural networks with many layers or recurrent networks often have extremely steep regions resembling cliffs. These result from the multiplication of several large weights together. On the face of an extremely steep cliff structure, the gradient can be become very large. Such points can be reached in small steps from flatter regions which would lead to large jumps in parameter space - to regions again far beyond the minimum.

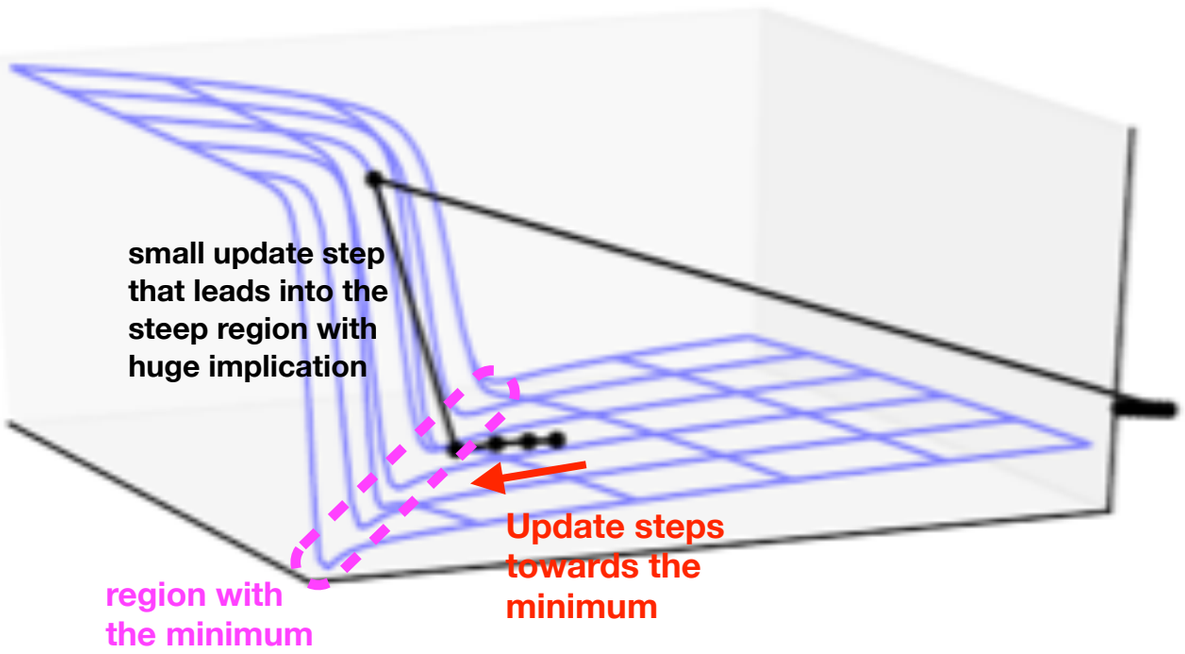
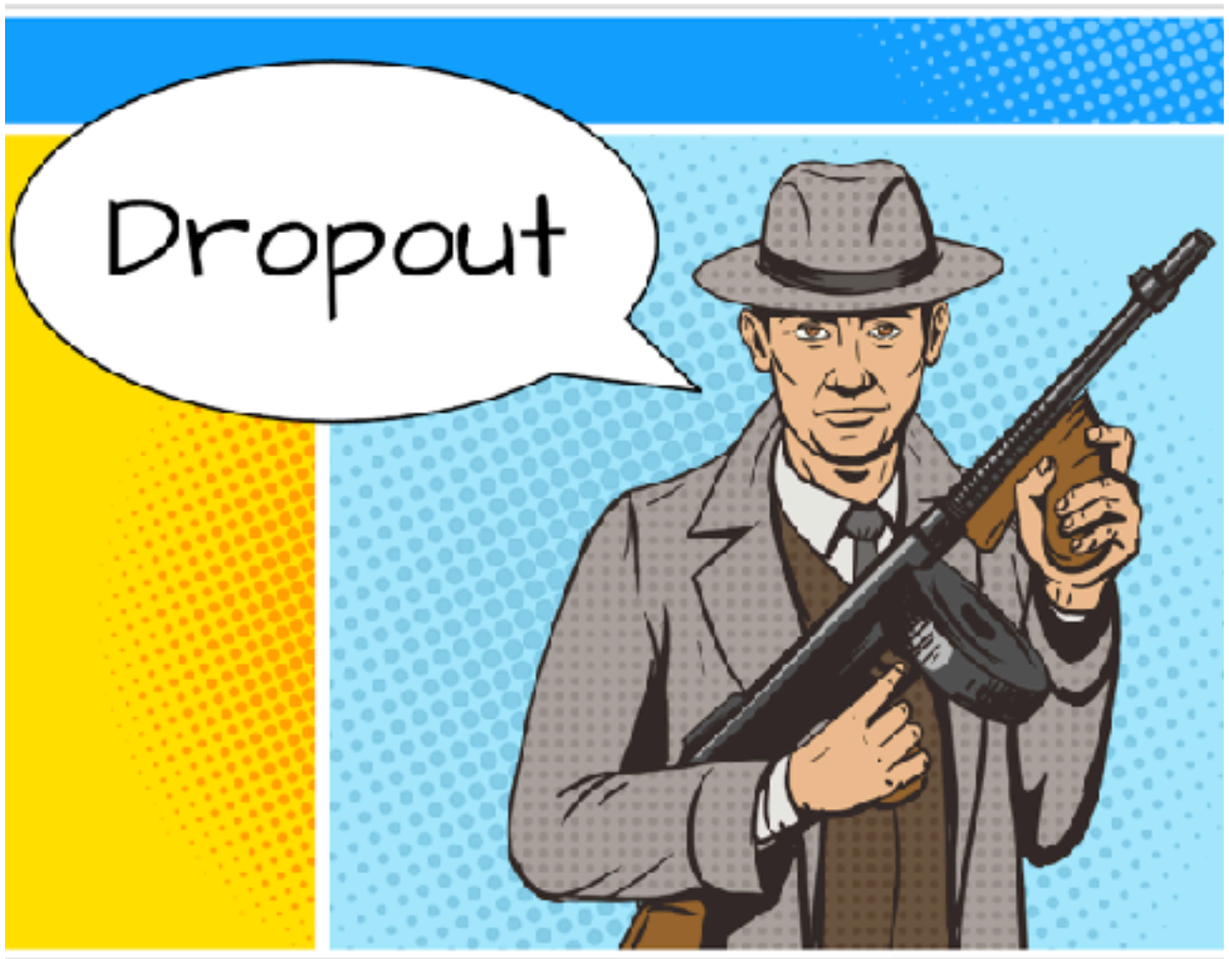


Figure from “Deep Learning”, I. Goodfellow et al.: Sharp nonlinearities in parameter space give rise to very high derivatives.

Regularisation



Regularisation as a Means to Avoid Overfitting

- Problem: Deep neural networks typically have a large number of parameters: ~10k - 1M. These networks can fit a huge variety of complex datasets — but bear the risk to overfitting the training set. With millions of parameters you can fit the whole zoo!

- Regularisation comes to the rescue!

- Goal: Learn the most popular regularisation techniques.

What is Regularisation

Well-posed problems (according to Hadamard)

- Problems that have a unique solution.
- Problems with a solution that continuously depends on the data.

⟷

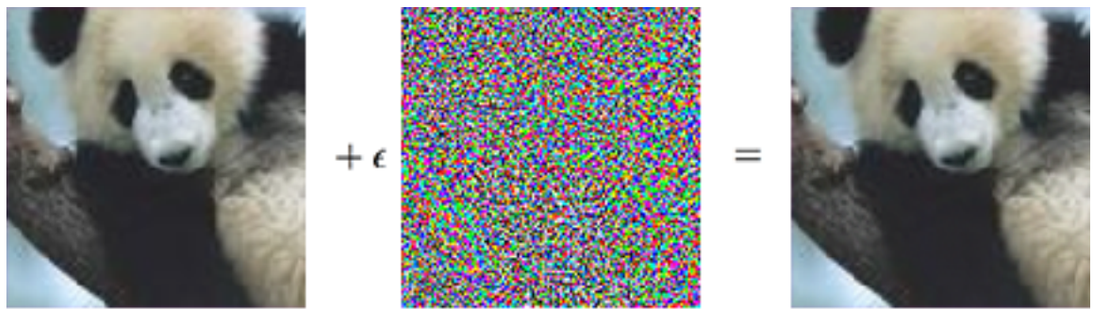
Ill-posed problems

- Opposite of well-posed problems.

Regularisation
A technique that attempts to turn ill-posed optimisation problems into well-posed problems.

Learning Algorithms Often Ill-Posed

- In machine learning, learning is formulated as optimisation problem — parameters are determined to best fit the training data. For typical models, this is ill-posed: By minimising the cost we easily get into the overfitting regime where the fitted model fails to generalise well.
- Another indicator for learning being ill-posed is the existence of adversarial examples:



see e.g. https://blog.openai.com/adversarial-example-research/ or I. Goodfellow, *Deep Learning*

Regularisation in the Light of Overfitting

- In machine learning, regularisation consists in modifying the learning algorithm such that preference is given to specific solutions over other solutions — e.g. by modifying the cost function.

- A typical objective is to tune and control model complexity — to give preference to simpler models and to avoid overfitting.
  - Consider family of polynomial models as hypothesis space. Find a good compromise between fitting well the training data and keeping the degree of polynomial sufficiently low to avoid overfitting.
  - Similarly, consider neural nets with L layers as hypothesis space. Give preference to models with smaller parameters or smaller number of parameters.

- Goodfellow et al:
  "*Regularisation is any modification to a learning algorithm that is intended to reduce its generalisation error but not its training error.*"

Regularisation Methods (Overview)

- Weight Decay (Weights Penalties)
  - Add constraints to the parameters to give preference to simple models — restriction in the length of the parameter vector or in number of parameters (sparsity).

- Dropout
  - Randomly drop neurons during training steps to make the solution less dependent on individual neurons.

- Early Stopping
  - Stop training at the minimum of the cost function on the validation set to avoid overfitting.

- Data Augmentation
  - Generate more training data to introduce additional characteristic features (e.g. symmetries) the solution should have.

Weight Decay

The loss function is modified to give preference to smaller or fewer weights — by adding a suitable penalty term.

$$J = J_0 + \lambda \Omega(\mathbf{W})$$

J_0 original loss function (e.g. RMS loss)

$\Omega(\mathbf{W})$ penalty term that favours models with smaller weights ← not a function of the activations of the network

$\lambda \geq 0$ regularisation parameter

Two forms of penalties are common:

$$L_1\text{-Regularisation: } \Omega(\mathbf{W}) = \|\mathbf{W}\|_1 = \sum_{l,k,j} \left| W_{kj}^{[l]} \right|$$

$$L_2\text{-Regularisation: } \Omega(\mathbf{W}) = \|\mathbf{W}\|_2^2 = \sum_{l,k,j} \left| W_{kj}^{[l]} \right|^2$$

For optimisation with gradient descent, the derivatives of the loss are modified by the penalty term in a direction to make the loss and the penalty term smaller.

Gradient Descent with L²-Regularisation

Gradient for the regularised loss function:

$$\nabla \left(J_0(\mathbf{W}) + \frac{\lambda}{2} \|\mathbf{W}\|^2 \right) = \nabla J_0(\mathbf{W}) + \lambda \mathbf{W}$$

This leads to the following update rule for gradient descent with L² regularised loss:

$$\begin{aligned} \mathbf{W} &\leftarrow \mathbf{W} - \alpha \nabla J(\mathbf{W}) \\ &\leftarrow \mathbf{W} - \alpha \left(\nabla J_0(\mathbf{W}) + \lambda \mathbf{W} \right) \\ &\leftarrow \underbrace{(1 - \alpha\lambda)}_{\text{reduction in length}} \mathbf{W} - \underbrace{\alpha \nabla J_0(\mathbf{W})}_{\text{original loss term}} \end{aligned}$$

The addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, just before performing the usual gradient update.

Gradient Descent with L^1-Regularisation

Gradient for the regularised loss function:

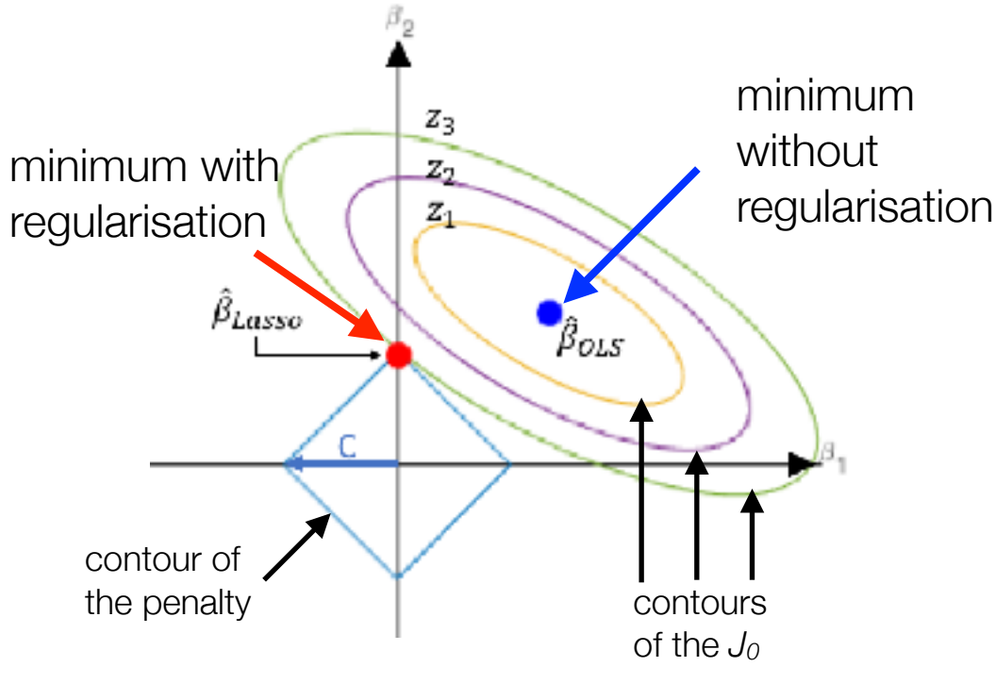
$$\nabla \left(J_0(\mathbf{W}) + \lambda \|\mathbf{W}\|_1 \right) = \nabla J_0(\mathbf{W}) + \lambda \, \mathrm{sign}(\mathbf{W})$$

element-wise application of sign-function

The effect on the update rule cannot be as clearly seen as for L^2 regularisation.

Generally, L^1-regularisation leads to sparser solutions than L^2 regularisation. Here, sparsity means that some parameters have an optimal value of zero. (For L^2-regularisation, the weights are just shrunk.)

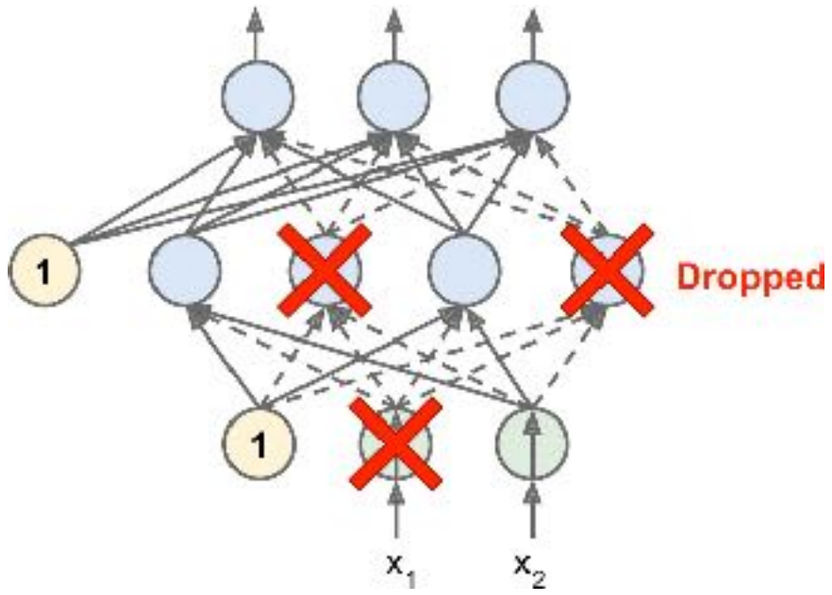
For linear regression problems the addition of the L^1 penalty term is also referred to as LASSO regression.



Dropout

- Most popular regularisation technique for deep neural networks.
- Proposed by G. E. Hinton in 2012 and further detailed in a paper by Nitish Srivastava et al.
- Highly successful: Even state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout.
- Algorithm:
  - At every training step, every neuron (including input neurons, excluding output neurons) has a probability *p* of being temporarily ignored during this training step —> masked activations.
  - The hyper-parameter *p* is called *dropout rate*, typically set to 50% for hidden, and 20% for input units.
  - After training for testing or in production, neurons don't get dropped, but weights or outputs need to be corrected since the weights have been trained including the dropout rate.

(1) G. Hinton et al., "Improving neural networks by preventing co-adaptation of feature detectors,"(2012)
(2) N. Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," (2014).



from A. Géron, *'Hands-On Machine Learning with Scikit-Learn and TensorFlow'*

Adapted from Andrew Ng. for the forward prop part:

```
import numpy as np

# a3 : activations in layer 3
d3 = np.random.rand(a3.shape[0],a3.shape[1]) < keep_prob
a3 = np.multiply(a3,d3)
a3 /= keep_prob
```

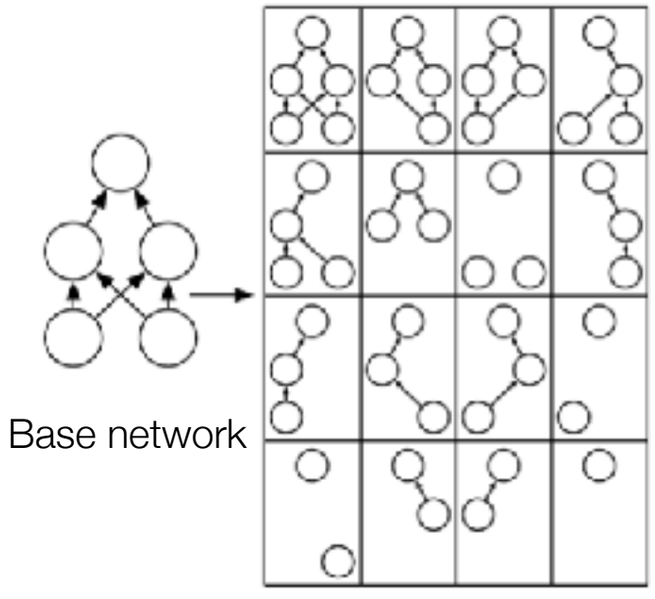
Intuition on Why Dropout Works

- Implication of using dropout:
  - Simpler models with less units are considered during training.
  - Trained model learns to be less dependent on single units and units cannot co-adapt with their neighbouring units; they have to be as useful as possible on their own.
  - A more robust network is obtained that generalises better.

From Geron, “Hands-On Machine Learning with Scikit-Learn and TensorFlow”:

“Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would obviously be forced to adapt its organization; it could not rely on any single person to fill in the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn’t make much of a difference. “

- Related to ensemble methods:
  Trains an ensemble of sub-networks derived from a base network by removing (non-output) units.



Implementation of Dropout

- Computationally very cheap:
  - Requires only O(n) computations per training step, per mini-batch: For each unit a random binary number (keep or drop unit)
  - O (n) additional memory to store these binary numbers for the backprop.
  - No additional cost at test time or in production.

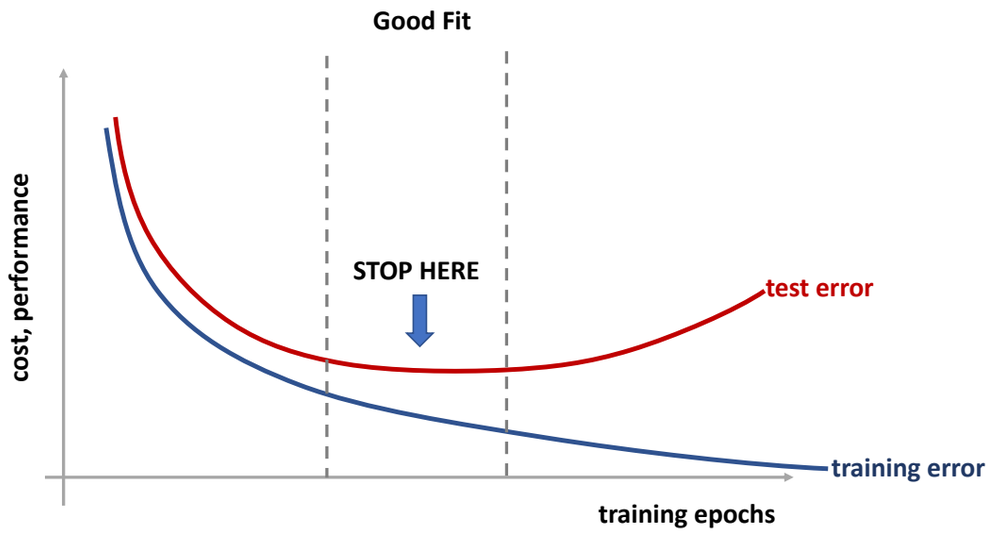
- Very versatile:
  - Does not significantly limit the type of model or training procedure that can be used. Applicable e.g. to MLP, CNN or RNNs or with SGD.
  - Can even be combined with other regularisation techniques.

- Reduced representational capacity:
  - As a regularisation technique, dropout reduces the effective capacity of a model — possibly, needs to be compensated with larger models. The training of these have an impact on performance - in the sense, training models with dropout is more costly.
  - For very large datasets, regularisation implies little reduction in generalisation error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularisation.

Early Stopping

- Often, the training error monotonically decreases while the validation error begins to increase after a certain number of epochs
  - A behaviour that can only be observed when training large models with sufficient representational capacity so that overfitting is possible.

- Algorithm
  - Run optimisation algorithm to train the model - simultaneously compute the validation set error.
  - Store a copy of the model parameters as long as the validation set error improves.
  - Iterate until validation set error stops improving (e.g. has not improved for *k* steps).
  - Return the parameters where the smallest validation set error is observed.



Early Stopping is such a simple and efficient regularisation technique that Geoffrey Hinton called it a "**beautiful free lunch**."

Why is Early Stopping a Regularisation Method?

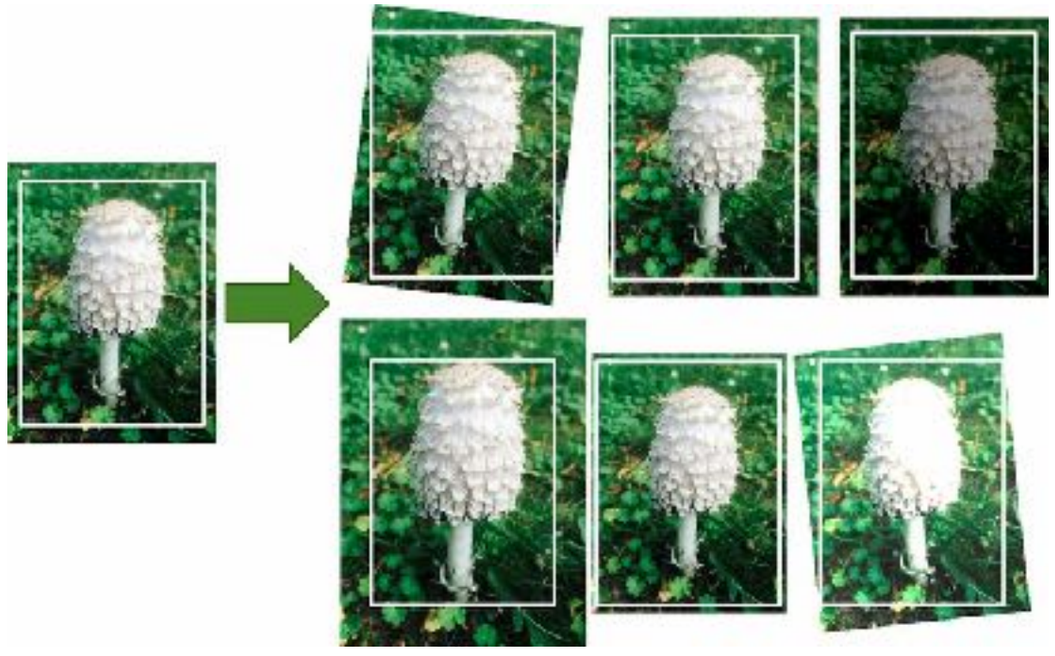
- Training time (number of training steps/epochs) considered as hyper-parameter.
- It controls the *effective capacity* of the model by determining the number of steps it can take to fit the training set.
  - Restricts the volume in parameter space to be searched: Assume gradient bounded by *C*, learning rate to be α and number of steps T; then, the effective volume in parameter space reachable from the initial parameters is $\|\theta - \theta_0\| \sim \alpha \cdot C \cdot T$.
  - In the case of a linear model with a quadratic error function and simple gradient descent, early stopping is equivalent to L^2-regularisation.(*)
- Efficient, non-intrusive search for this hyper-parameter:
  - The cost relates to repeated calculations of the validation set error and keeping a copy of the recent model parameters. The first can easily be parallelised.
  - Almost no change to the code needed.
- Can easily be combined with other regularisation techniques.
  - Even when using regularisation strategies that modify the objective function to encourage better generalisation (weight decay), it is rare for the best generalisation to occur at a local minimum of the training objective.

(*) See Goodfellow et al.,“Deep Learning”.

Data Augmentation

Means to counter overfitting:

- Reduce model complexity: Explicitly by choosing simpler models, or by using a regularisation scheme that effectively reduces the model complexity.
- Increase the amount of training data: Collecting new data or generating new samples from exiting ones.



from A. Géron, *'Hands-On Machine Learning with Scikit-Learn and TensorFlow'*

- Generate new realistic samples from existing ones. Modifications applied to the data should be learnable.
- Modifications, e.g. in a computer vision task: Shifting, rotating, flipping, resizing, varying contrast, brightness, saturation by various amounts.
- By training with data that includes these modifications, the model is forced to be more tolerant to the position, orientation, size, contrast, etc. — see an object recognition where the inputs are high-dimensional with many factors of variations the classification should be invariant to.
- Modified samples often generated on the fly.

Applying Data Augmentation

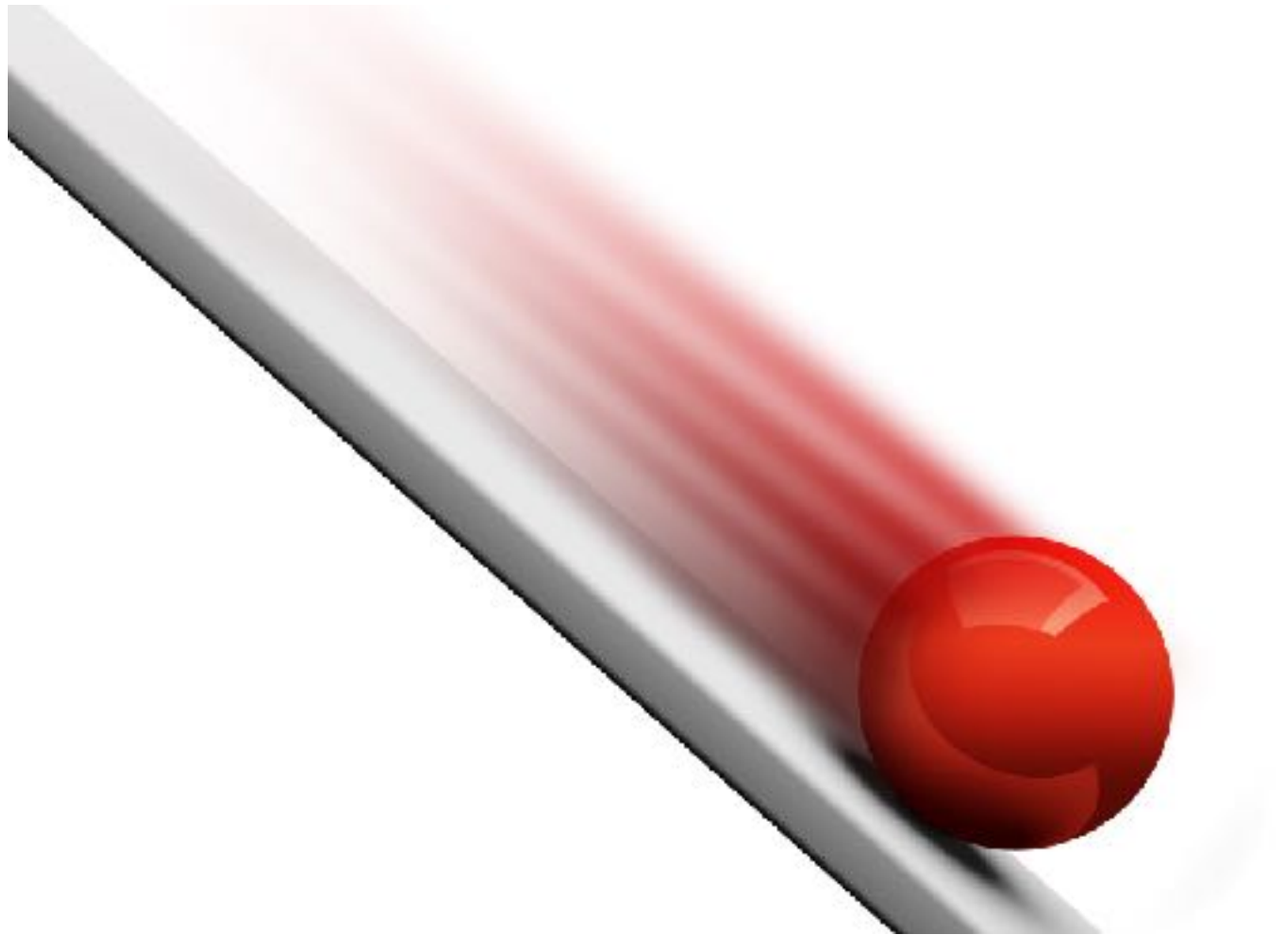
- Typically used for classification tasks.

- Be careful so that no transformations are applied that would change the correct class.
  - Example: In optical character recognition, the model should be able to differentiate between ‘b’ and ‘d’ or ‘6’ and ‘9’. Therefore, flips and 180◦ rotations are not valid transformations for augmenting the datasets here.

- May be considered rather as preprocessing step. However:
  - Since it helps reducing overfitting we here list it as a regularisation method.
  - It is often preferable to generate training instances on the fly during training rather than wasting storage space and network bandwidth.

- Some libraries provide good support for data augmentation.
  - TensorFlow offers several image manipulation operations such as transposing (shifting), rotating, resizing, flipping, and cropping, as well as adjusting the brightness, contrast, saturation, and hue.

Advanced Optimisation Methods



Momentum

RMSprop

Adam

Gradient Descent in Deep Learning Models

- Gradient Descent (BGD, MBGD, SGD):
  - Follow trajectory that leads to lower cost values.
  - Typically works fine for convex functions: There is always a way to move down until you reach the minimum (with MBGD or SGD this is not guaranteed, though).
  - Possibly, learning may get very slow in flat regions.
- Cost functions in deep NN models are non-convex: Danger to get stuck in points where the gradients get very small or zero.
  - Local, Non-Global Minima
  - Saddle Points
  - Flat Regions
- Non-global minima are *not* considered as an issue:
  - Typical network structures have symmetries (e.g. an MLP w.r.t. permutations of the units in a layer and the according parameters). An according number of global minima must exist.
  - Other non-global minima are considered to be very rare: Cost function needs to grow in all directions — a condition hard to meet for function in high dimensional spaces.
  - BGD and SGD can help to escape from local minima or saddle points.
- With saddle points or flat regions learning can get very slow.
  - Some improvements beyond gradient descent are desirable. —> Faster optimisers.

Overview on Faster Optimisers

Momentum, Nesterov
Allows to surpass flat regions or saddle points — like a ball that keeps on rolling down if it has an initial speed when entering flat regions.

RMS Prop
Adaptively adjusts the learning rate to incorporate differences in the steepness along different directions in parameter space.

Adam
Combination of Momentum / Nesterov and RMSprop

State of the Art is using Adam.

Momentum, Nesterov

- Modified scheme for computing the gradient.
- Intuition (from A. Géron, 'Hands-On Machine Learning with Scikit-Learn and TensorFlow')

> "*Imagine a ball rolling down a gentle slope on a smooth surface. It starts out slowly but quickly picks up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance).(…) In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom.*"

- Algorithm: Compute an exponentially decaying moving average of past gradients and move in the direction of the moving average.

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + \alpha \nabla J(\theta)$$
$$\theta \leftarrow \theta - \mathbf{m}$$

'momentum' hyper-parameter which controls the decay and the friction (large friction and fast decay) with small values of β_1 and vice versa. Good default: $\beta_1 = 0.9$

Actually not evaluated on the whole training set but just on mini-batches.

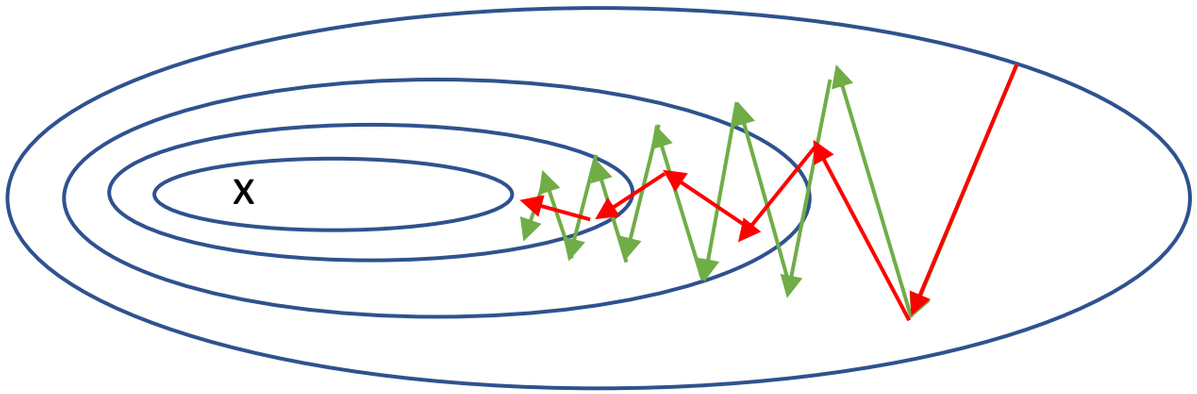
With **Nesterov**, this term is modified to

$$\alpha \nabla J(\theta + \beta_1 \mathbf{m})$$

which is considered to be more efficient.

Momentum, Nesterov (2)

Typical Learning Trajectories in Parameter Space (w or w/o momentum)

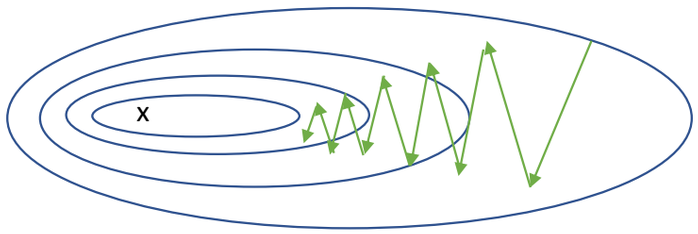


As compared with original gradient descent (green), the trajectory with momentum more directly approaches the minimum at ‘x’. Due to the moving average, the ‘transverse’ component of the gradient is more and more eliminated.

Implementation affects just the parameter update rule
(not the model’s forward prop and backprop mechanics).

RMS Prop

- Idea: Adjust the learning rate (adaptively)
  - Increase learning rate in direction of slow progress (small components in gradient vector).
  - Decrease learning rate in direction of fast progress (large components in gradient vector).



Large component of gradient, smaller learning rate desirable.

Small component of gradient, speedup, large learning rate desirable.

Homogenise learning speed in the different directions.

- Algorithm:
  - For each component of the gradient vector, build an exponentially decaying measure with the typical scale of the components of the gradient vector.
  - Divide each component in the gradient by the scale.

$$\begin{aligned} \mathbf{s} &\leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla J(\theta) \odot \nabla J(\theta) \\ \theta &\leftarrow \theta - \frac{\alpha}{\sqrt{\mathbf{s}+\epsilon}} \odot \nabla J(\theta) \end{aligned}$$

element-wise operations

Implementation affects just the parameter update rule

Good default for the decay parameter β_2 : $\beta_2 = 0.999$

Adam Optimisation

- Combines Momentum and RMS Prop and is currently considered as the de facto standard.

- Algorithm:

$$
\begin{aligned}
\mathbf{m} &\leftarrow \beta_1 \mathbf{m} + (1-\beta_1)\nabla J(\theta) \\
\mathbf{s} &\leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla J(\theta) \odot \nabla J(\theta) \\
\hat{\mathbf{m}} &= \frac{\mathbf{m}}{1-\beta_1} \\
\hat{\mathbf{s}} &= \frac{\mathbf{s}}{1-\beta_2} \\
\theta &\leftarrow \theta - \alpha \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{s}}+\epsilon}} \odot \nabla J(\theta)
\end{aligned}
\quad \Big\} \ (*)
$$

- Implementation affects just the parameter update rule
- Element-wise operations are adopted for the squared gradient, the division and square-root in the update step.
- Initialise ***m*** and ***s*** to 0. In order to avoid that at the beginning these terms are systematically smaller in magnitude, the steps (*) are applied.

- Hyper-Parameters:

Learning Rate	α	0.001
Momentum	β_1	0.9
RMS Decay	β_2	0.999
Numerical Stabilisation	ϵ	1.0E-08

Learning Schedule

- Instead of adapting the learning rate in accordance with information collected during the optimisation process along the trajectory in parameter space we may manually adjust the learning rate in the different phases of learning (~ epochs or update steps).
- Typically, start with high learning rate, then reduce it:

Piecewise Constant Learning

Epochs	Rate
1-100	0.1
101-300	0.01
301-500	

Performance scheduling
Measure the validation error every *N* steps, reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling
Set the learning rate to a function of the iteration number *t*:

$$\alpha(t) = \alpha_0 \cdot e^{-t/T}$$

Power Scheduling

$$\alpha(t) = \alpha_0 (1 + t/T)^{-c}$$

The hyperparameter c is typically set to 1. Similar to exponential scheduling, but learning rate drops much more slowly.