Deep Learning:

Training Algorithms

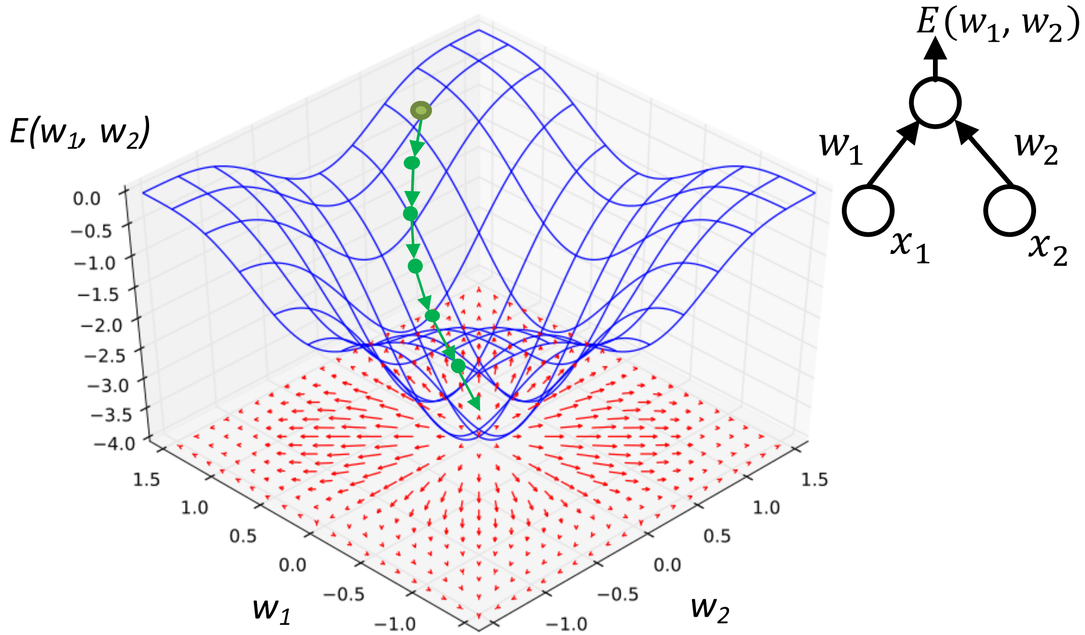
Ozan Özdenizci
Institute of Theoretical Computer Science
ozan.ozdenizci@igi.tugraz.at

Deep Learning VO - WS 23/24
Lecture 5 - November 6th, 2023

Recap: Stochastic Gradient Descent (SGD)

- The **gradient** points in the direction of the steepest increase of the error.

$$\nabla_{\mathbf{w}} E = \begin{pmatrix} \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_D} \end{pmatrix}$$



- We slightly change parameters to reduce error:

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \epsilon \nabla_{\mathbf{w}} E(\mathbf{w}_{old})$$

$\epsilon > 0$ is the **learning rate.**

- **Stochastic Gradient Descent:**
  - Divide the training set into **mini-batches**.
  - Approximate the true gradient by the gradient over the mini-batch.

```
1 Choose w randomly
2 Let B be the set of minibatches
2 i ← 0
3 REPEAT
4     FOR b in B
4           w ← w − ϵ(i) ∇w E(w, b)
5           i ← i+1
6 UNTIL Stopping Criterion
```

Today

- [ ] Basic Training Algorithms
- [ ] Common Pitfalls & Training Considerations
- [ ] Searching Hyperparameters

(1) SGD with decreasing learning rate

Due to the batch-induced noise in stochastic GD, gradient does not become 0 at minimum.

- One therefore reduces the learning rate ϵ_k for each update k.
- Sufficient condition to guarantee convergence of SGD:

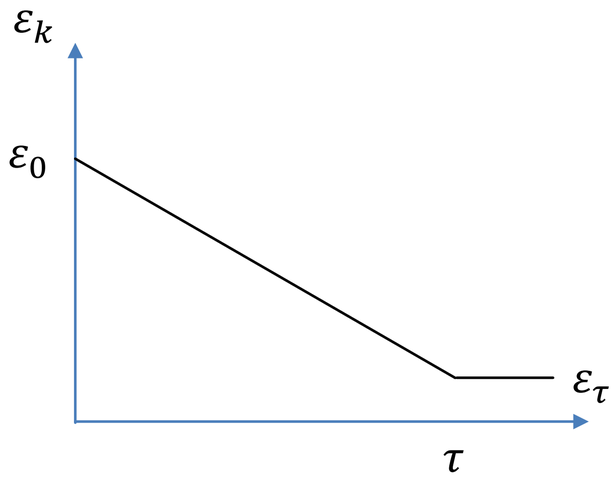
$$\sum_{k=0}^{\infty} \epsilon_k = \infty \quad \text{and} \quad \sum_{k=0}^{\infty} \epsilon_k^2 < \infty$$

A commonly used approach (in practice):

- Start with initial rate ϵ_0
- Decrease linearly until update τ

$$\epsilon_k = \left(1 - \frac{k}{\tau}\right) \epsilon_0 + \frac{k}{\tau} \epsilon_\tau$$

- Then keep constant ϵ_τ



Choosing the meta-parameters:

τ : A few passes through the training examples (e.g., 10 epochs)

ϵ_τ : about 1% of ϵ_0

ϵ_0 : Monitor first several iterations and

choose learning rate higher than the best performing,

but not so high so that it does not cause severe instability.

(2) Momentum

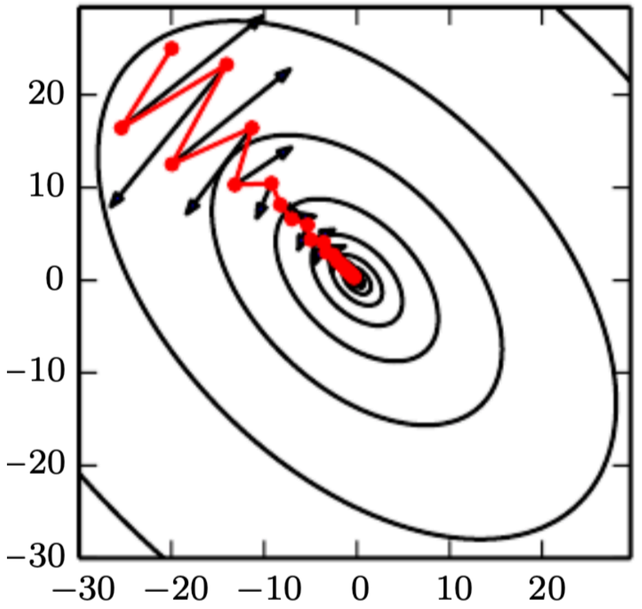
Use a running average of the gradient for the update:

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_\theta L(\theta)$$

$$\theta \leftarrow \theta + \mathbf{v}$$

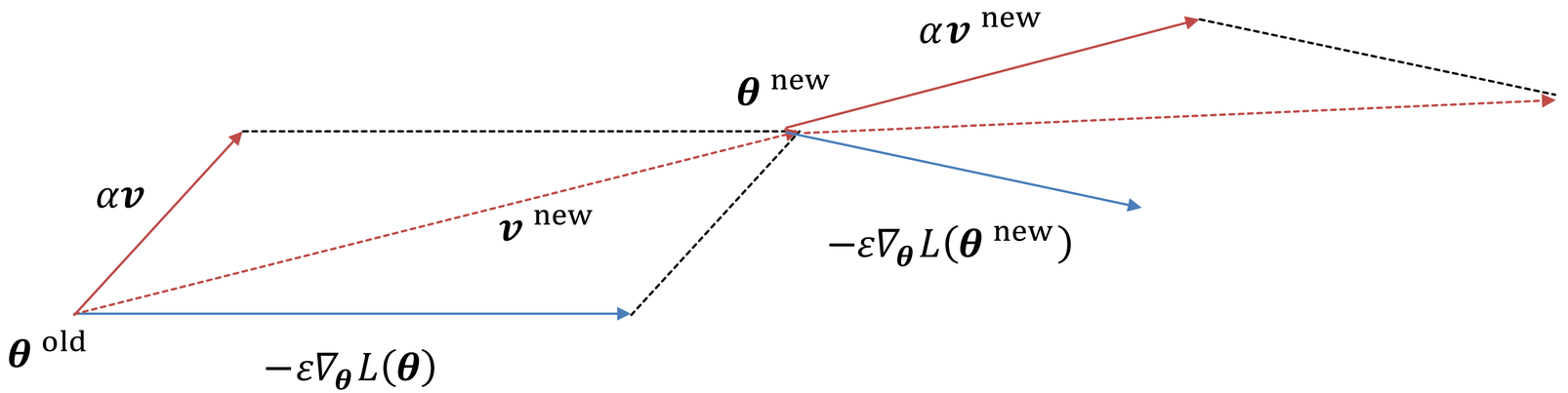
$\mathbf{v}$: velocity

L : objective function (loss)



Basic idea: Give the parameter vector a **mass** that moves on the error landscape.

- Reduces gradient noise of minibatches
- Reinforces consistent gradient directions



Choosing the meta-parameters:

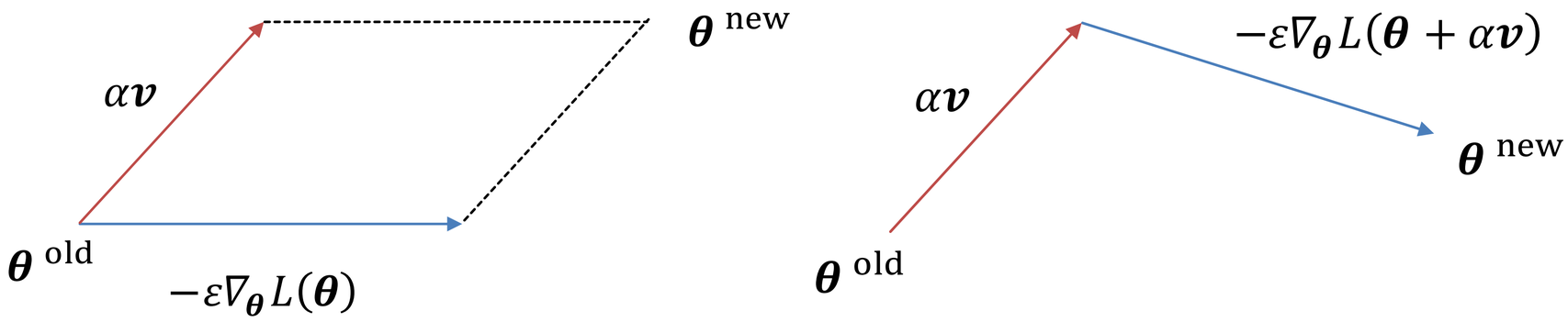
$\alpha \in [0,1)$: How quickly previous contributions decay (common choices: 0.5, 0.9, 0.99).

Often adapted over time, starting with small value and then increased.

(3) Nesterov Momentum

Evaluate at the location where parameters would be if you continued with velocity (no gradient force).

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_\theta L(\theta + \alpha\mathbf{v})$$
$$\theta \leftarrow \theta + \mathbf{v}$$



- The Nesterov-Momentum variant has stronger theoretical convergence properties on convex functions and slightly better results in practice.

Sutskever, Ilya, et al. "On the importance of initialization and momentum in deep learning." ICML 2013.

(4) Algorithms with Adaptive Learning Rates

The error is often highly sensitive to some directions in parameter space and insensitive to others.

- Hence, adapting a *global learning rate* is less helpful.
- Instead, one often tries to find **individual learning rates** for each parameter.

Algorithms:

- (4.1) AdaGrad
- (4.2) RMSProp
- (4.3) Adam

(4.1) AdaGrad

"Adaptive Gradients"

Scale learning rates by dividing them by the square root of the sum of all historical squared 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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 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: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
    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

(4.1) AdaGrad

"Adaptive Gradients"

Scale learning rates by dividing them by the square root of the sum of all historical squared 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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 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: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root a
element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
end while

- This is quite useful for convex optimization.
- *Limitation:* For non-convex optimization, early gradients (accumulated at the beginning of training) are often not relevant later.

AdaGrad - A brief summary:

$$\mathbf{g} = \begin{pmatrix} \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_D} \end{pmatrix}$$

Gradient computed over one mini batch.

$$\mathbf{r} = \begin{pmatrix} \sum_j \left(\frac{\partial L^{(j)}}{\partial w_1} \right)^2 \\ \vdots \\ \sum_j \left(\frac{\partial L^{(j)}}{\partial w_D} \right)^2 \end{pmatrix}$$

Squared gradients accumulated across all minibatches so far

$$\Delta w_k = -\frac{\epsilon}{\delta + \sqrt{\sum_j \left(\frac{\partial L^{(j)}}{\partial w_k} \right)^2}} \cdot \frac{\partial L}{\partial w_k}$$

individual update for a single parameter w_k

(4.2) RMSProp

"Root Mean Squared Propagation"

Like AdaGrad, but using a ***running average*** to compute the square root of the sum of historical squared gradients.

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

(4.2) RMSProp

"Root Mean Squared Propagation"

Like AdaGrad, but using a *running average* to compute the square root of the sum of historical squared gradients.

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

RMSProp

- Does not shrink the learning rate according to the entire history of the squared gradients.
- Instead, uses a decaying average to discard history from the extreme past.

(4.2) RMSProp with Nesterov Momentum

RMSProp can also be combined with the idea of momentum.

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

Initialize accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

(4.3) Adam

"Adaptive Moments"

Similar to RMSProp with momentum, but with corrected moments.

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 $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  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

Adam

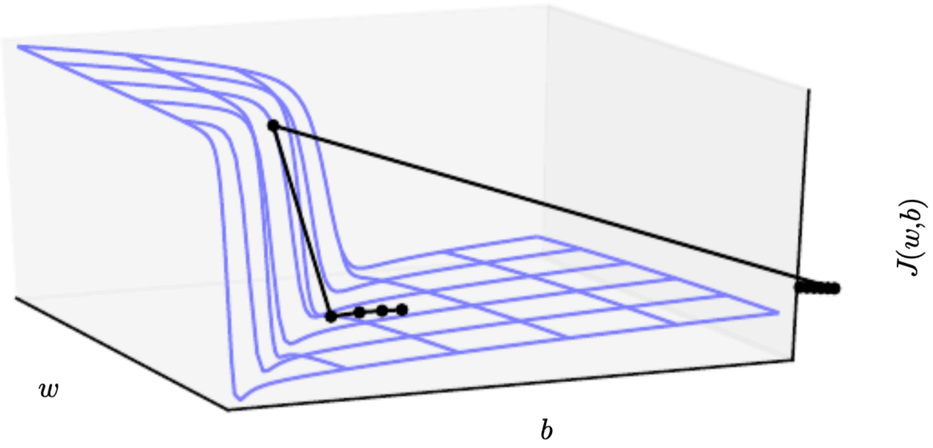
- Initially, the moments are biased towards the **initialization**. This is corrected in Adam.
- Compared to SGD, Adam requires much less hyper-parameter tuning. Default hyper-parameters often work well (in practice).

Today

- [x] Basic Training Algorithms
- [ ] Common Pitfalls & Training Considerations
- [ ] Searching Hyperparameters

Gradient Clipping

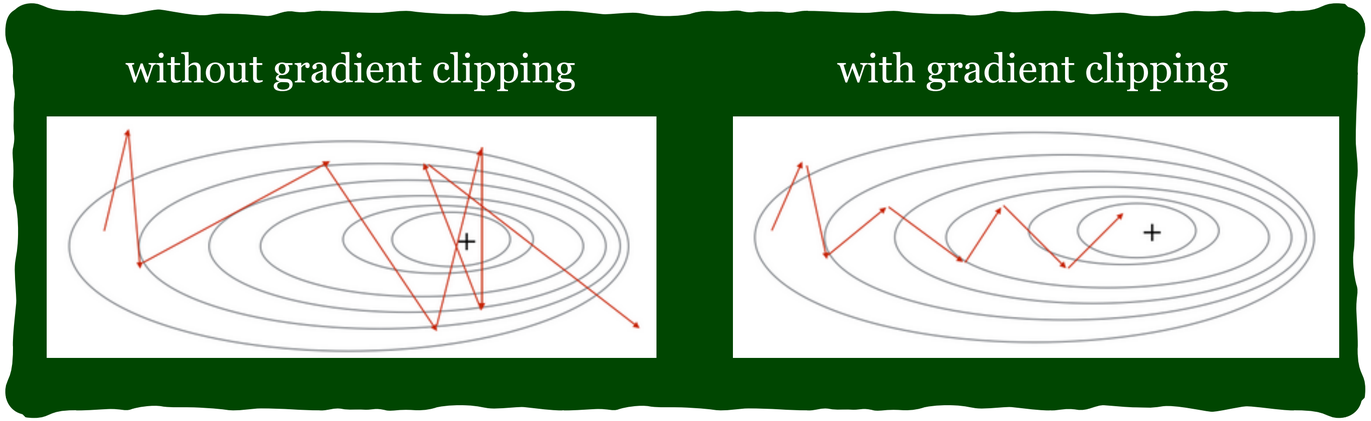
Deep networks often have extremely steep regions in the error surface, resembling to cliffs.



There, the gradient can become very large, which leads to instabilities and to large jumps in parameter space.

Gradient clipping: Clip large gradients by a threshold.

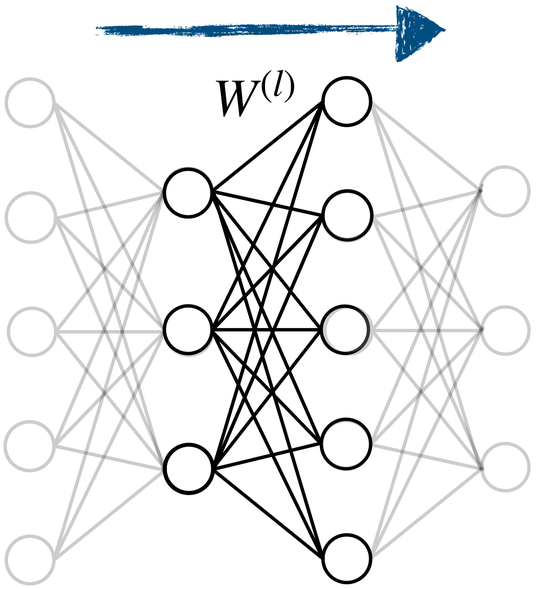
$$\text{if} \quad ||\mathbf{g}|| > v \quad \text{then} \quad \mathbf{g} \leftarrow \frac{v\mathbf{g}}{||\mathbf{g}||}$$



Parameter Initialization

Initial parameters can determine whether learning converges at all, or how quickly it converges.

Important considerations for network <u>weights</u>:



- **Breaking the symmetry:**
  - Units should compute different functions.
  - Initialize weights randomly (e.g., using a Gaussian or uniform distribution).
- **Sizes of initial weights:**
  - Large
    - Stronger symmetry breaking and avoids losing the signal.
  - Too large
    - Exploding values during forward/backward pass, or saturated activation functions (bad).
  - Small
    - Regularization: We want the weights to be small, so start with small ones.

Parameter Initialization - Heuristics

Consider the weights of a fully connected layer with m inputs and n outputs.

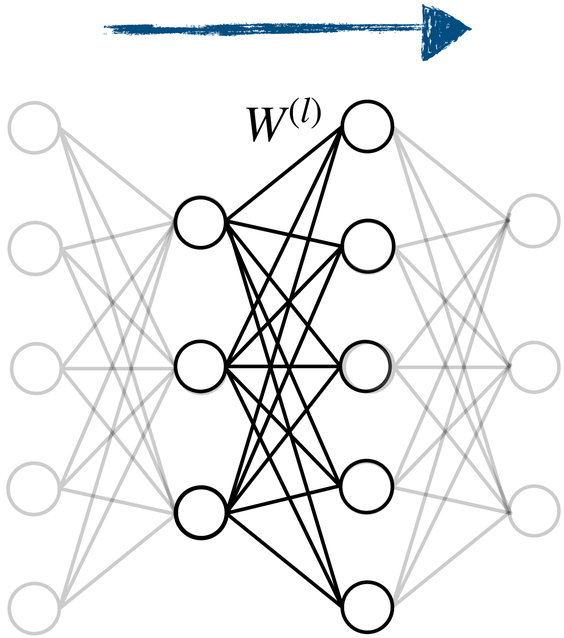
Heuristic 1:

$$w_{ij} \sim \text{Uniform}\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

Heuristic 2:

$$w_{ij} \sim \text{Uniform}\left(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right)$$

[Glorot & Bengio, AISTATS 2010]



$W^{(l)}$ is an $n \times m$ matrix

m : inputs

n : outputs

Potential problem for large layers: Each individual weight becomes very small.

Alternative: **Sparse initialization** - each unit has initially exactly *k* non-zero weights.

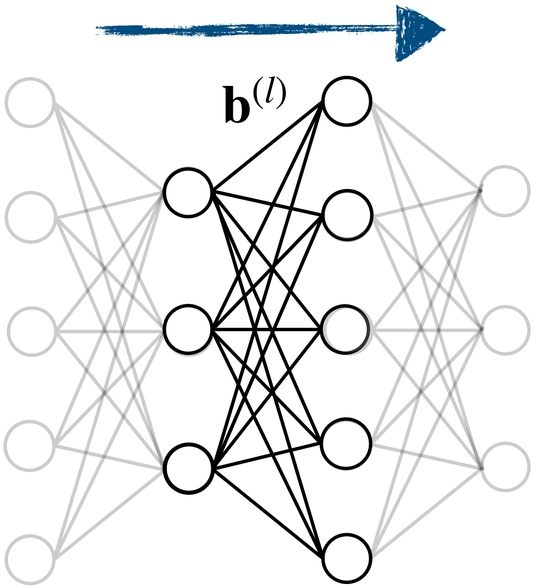
Empirical: Look at distribution of outputs and gradients on a single minibatch.

Parameter Initialization

Initial parameters can determine whether learning converges at all, or how quickly it converges.

Important considerations for network <u>biases</u>:

- Can all be set to the same heuristically chosen constant.
- For the output, one can choose biases to obtain right target means.
- Avoid saturation
  - e.g., ReLUs make activations mostly positive (small positive bias, e.g., 0.01)



Further techniques:

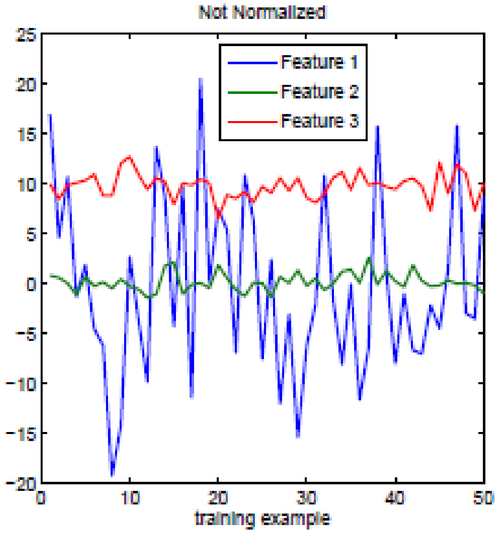
- Unsupervised pre-training
- Pre-training on similar tasks (e.g., for image-related tasks one can use networks pre-trained on the ImageNet dataset, which will already capture many aspects of natural images)

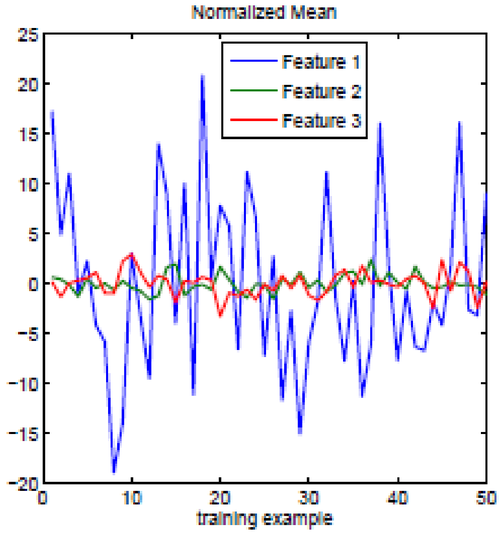
Activity Normalization Methods

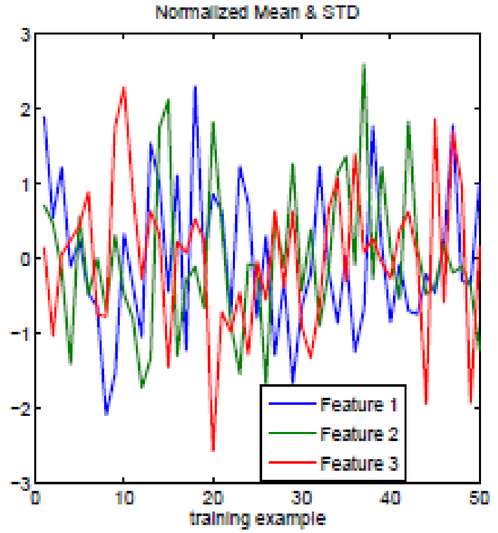
Input normalization

- We usually normalize each input feature to have zero-mean and unit-variance.
- This can significantly speed up learning.
- Mean and std. statistics are computed over the training set, and then applied to training, validation and test sets.

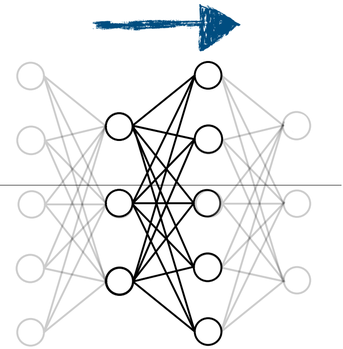
$$\tilde{x}_i^{(n)} = \frac{x_i^{(n)} - \text{mean}\left(\{x_i^{(1)}, \ldots, x_i^{(N)}\}\right)}{\text{std}\left(\{x_i^{(1)}, \ldots, x_i^{(N)}\}\right)}$$







Batch Normalization



We can interpret the output of a layer as the input to the next layer.

In Batch-Normalization, one does this normalization on the output of each layer for the current batch.

$$\tilde{z}_i^{(n)} = \frac{z_i^{(n)} - \text{mean}\left(\{z_i^{(1)}, \ldots, z_i^{(N')}\}\right)}{\sqrt{\text{var}\left(\{z_i^{(1)}, \ldots, z_i^{(N')}\}\right) + \epsilon}}$$

$z_i^{(1)}, \ldots, z_i^{(N')}$: output of unit i in the batch

ϵ : small constant to avoid division by zero

- This can be seen as an intermediate layer that linearly transforms the layers' output.
- During gradient descent, the error is backpropagated through this transformation.
- **When testing a single example**, one can use an average of the means and standard deviations in a number of previous batches.

Batch Normalization

$$\tilde{z}_i^{(n)} = \frac{z_i^{(n)} - \text{mean}\left(\{z_i^{(1)}, \ldots, z_i^{(N')}\}\right)}{\sqrt{\text{var}\left(\{z_i^{(1)}, \ldots, z_i^{(N')}\}\right) + \epsilon}}$$

$z_i^{(1)}, \ldots, z_i^{(N')}$: output of unit i in the batch

ϵ : small constant to avoid division by zero

Normalizing the output of units can reduce the expressive power of the network.

- Using trained (learnable) parameters γ_i , β_i , we often replace $\tilde{z}_i^{(n)}$ as:

$$\gamma_i \tilde{z}_i^{(n)} + \beta_i$$

- The resulting network is equivalent to the original one (without batch normalization).
- However, it has better learning dynamics. The mean and std. dev. of a unit's output is now determined by two parameters independent of other network parameters.
- Batch-Normalization is not applicable to small batch sizes and recurrent neural networks.

Layer Normalization

$$\tilde{z}_i^{(n)} = \frac{z_i^{(n)} - \text{mean}\left(\{z_1^{(n)}, \ldots, z_m^{(n)}\}\right)}{\sqrt{\text{var}\left(\{z_1^{(n)}, \ldots, z_m^{(n)}\}\right) + \epsilon}}$$

Here, we normalize over all outputs of a layer.

$z_1^{(n)}, \ldots, z_m^{(n)}$: outputs of all units of one layer for training example n

- This is not just a reparametrization. Every example will have its individual transform.
- Can be computed online, also with batch-size 1.
- Works also easily for recurrent neural networks.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer normalization." *arXiv preprint arXiv:1607.06450* (2016).

Today

- ☑ Basic Training Algorithms
- ☑ Common Pitfalls & Training Considerations
- ☐ Searching Hyperparameters

Hyperparameter search

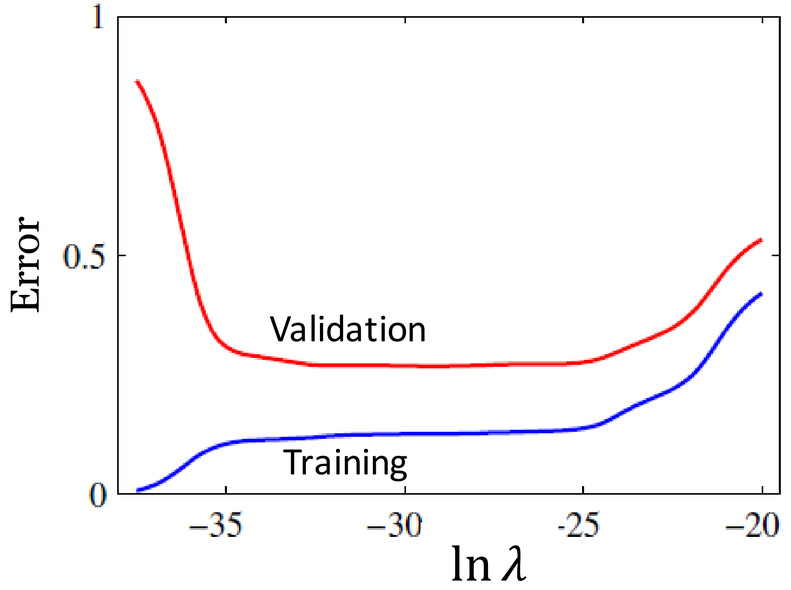
For each application, a set of hyperparameters (h-parameters) has to be defined, such as:

- Network architecture,
- Learning rate, number of iterations, batch size,
- Learning algorithm and its parameters (e.g. momentum),
- Regularization hyperparameters (*we will see next lecture!*)

Validation

- Performances of different h-parameter settings are compared on a **validation set**.
- When the best setting was determined, one can train the model with this set included in the training data.
- *Example:* Tuning the weight decay regularization parameter:

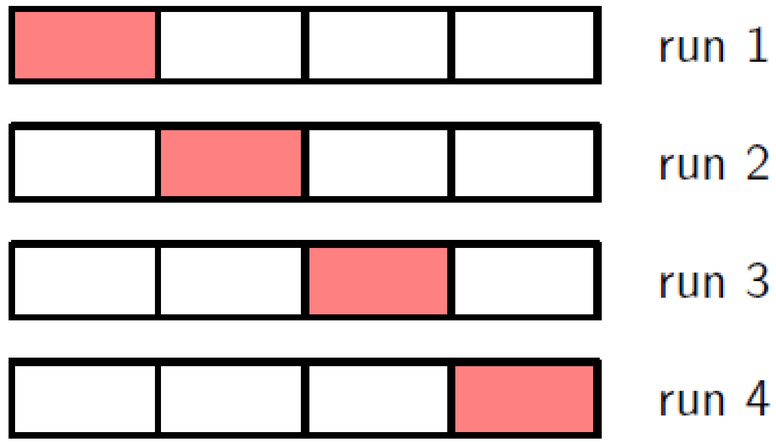
$$E(\mathbf{w}) = \sum_{n=1}^{N} \{\hat{y}(x^{(n)}) - t^{(n)}\}^2 + \lambda ||\mathbf{w}||^2$$



Cross-validation

Problem with small data sets:

- When validation set is small, we get a bad estimate of generalization error.
- When training set is small, the model is not well-trained.
- For small data sets, one can use *m-fold cross validation*.



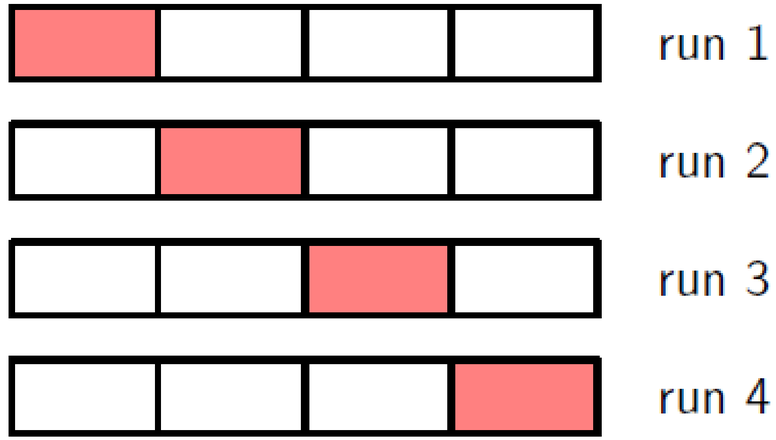
<u>m-fold Cross Validation:</u>

- Divide data $\mathscr{D}$ into (roughly) equally sized subsets $\mathscr{D}_1, \mathscr{D}_2, \ldots, \mathscr{D}_m$
- For $i = 1$ TO m
  - Training set is given by $\mathscr{D}_1 \cup \ldots \cup \mathscr{D}_{i-1} \cup \mathscr{D}_{i+1} \cup \ldots \cup \mathscr{D}_m$
  - Validation set is given by $\mathscr{D}_i$
- One obtains m validation errors. Compute mean and variance of those.

Cross-validation

Example application: Determine regularization parameter λ

- Make one CV run for each of a set of reasonable λ values
- Take the λ with the best average validation error
- Retrain the model with this λ and the whole data (excluding test set)
- Get an estimate generalization error using the test set.



Leave-one-out CV: Special case where m is equal to the size of the data set.

Stratified CV: Distribute the data on the folds such that targets (labels) have approximately the same distribution.

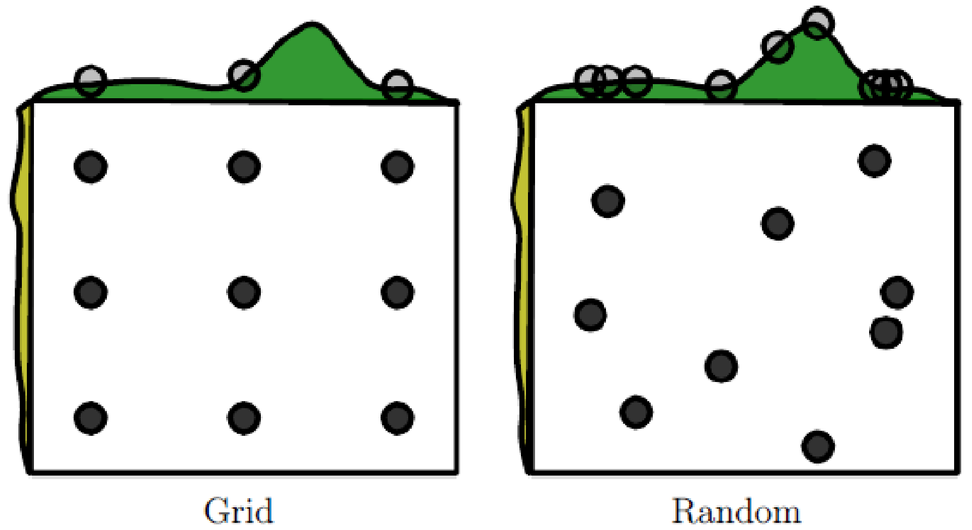
Options for Hyperparameter search

- **Grid-search** (test many h-parameter settings)
- **Random search** (improves on Grid-search)
- **Neural Architecture Search (NAS)** (automating architecture search)
- **Manual tuning** (often used)

Grid-Search & Random Search

Grid-search

- Define a grid on h-parameters values, test each point.



Random search

- Define interval and distribution for each h-parameter.
- Draw each h-parameter independently.
- Train model with h-parameter setting.
- Advantages: Tests more values for each individual h-parameter.
- Often used distributions: Uniform or uniform on log-scale.

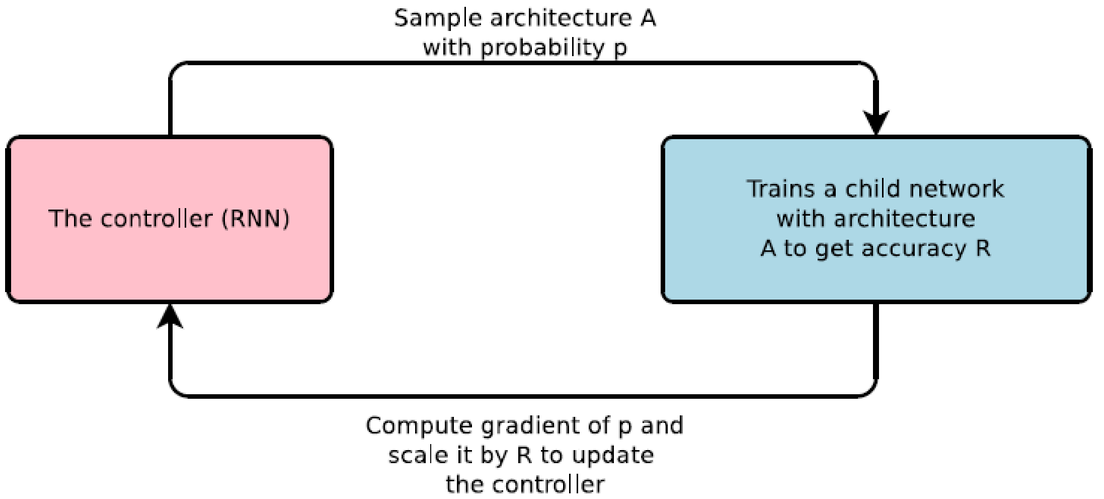
If the data set is small, one can still use m-fold cross validation for each point.

Neural Architecture Search (NAS)

- A technique for automating the design of neural networks.
- Resulting networks are on par or outperform hand-designed architectures.

General idea:

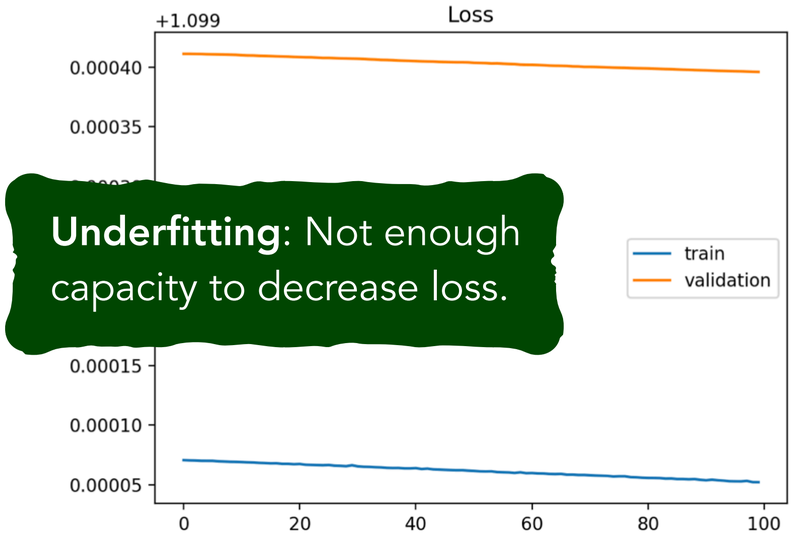
- Use a recurrent neural network (RNN, the controller) that samples network architectures.
- Performance is evaluated, better performance leads to larger reward
- The RNN is optimized with reinforcement learning to maximize the expected reward
- Disadvantage: Very compute-hungry



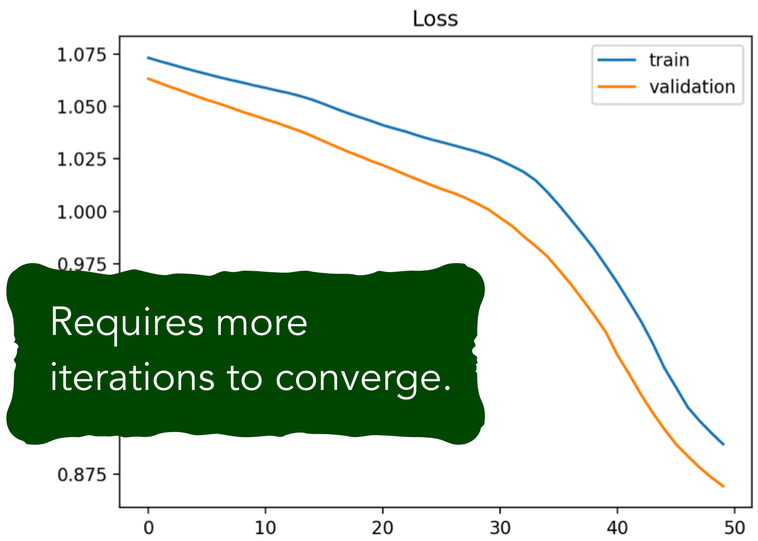
Zoph, B., & Le, Q. V. (2016). "Neural architecture search with reinforcement learning." *arXiv preprint arXiv:1611.01578*.

Monitoring the Training Process

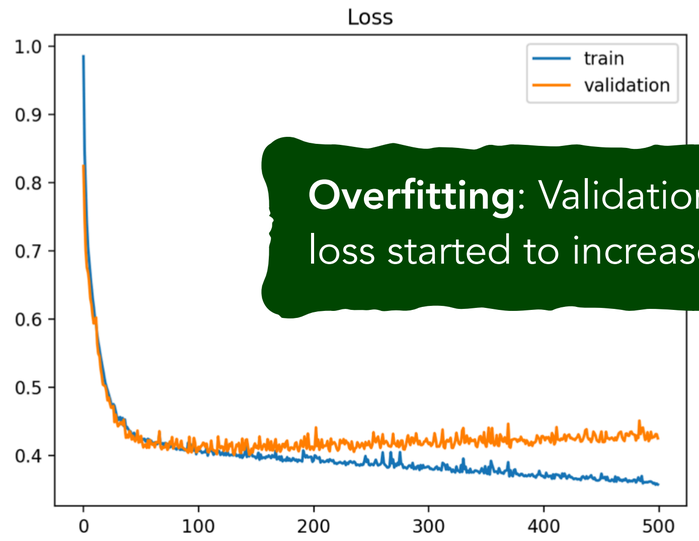
slide figures from
notes by A. Geiger



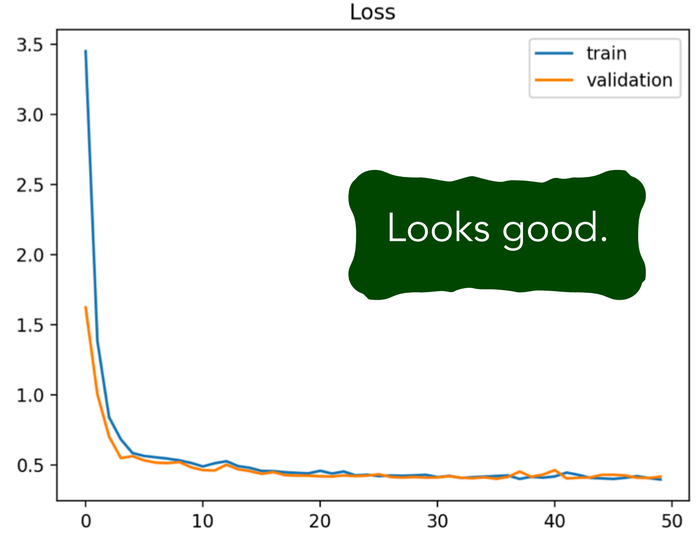
Underfitting: Not enough capacity to decrease loss.



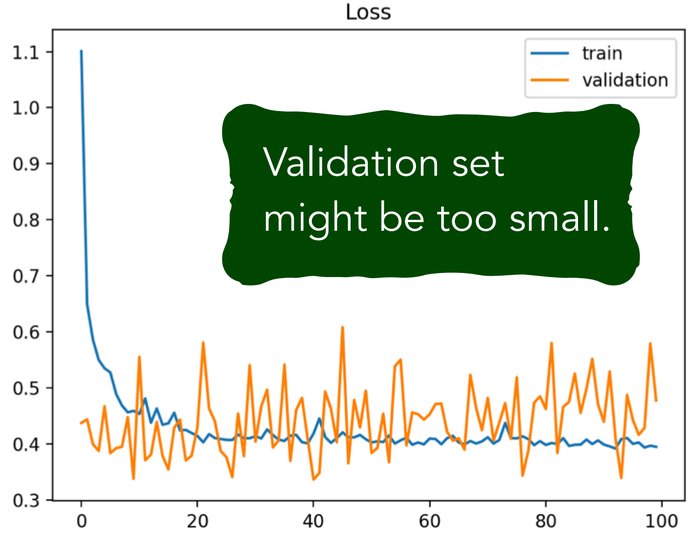
Requires more iterations to converge.



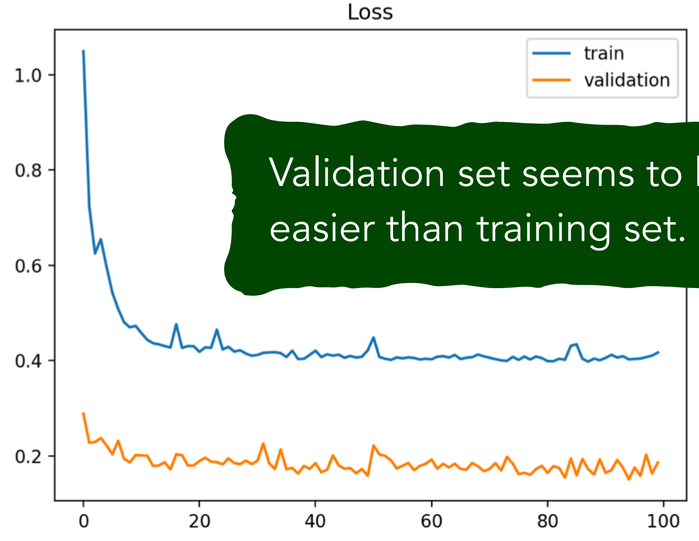
Overfitting: Validation loss started to increase.



Looks good.



Validation set might be too small.



Validation set seems to be easier than training set.

Today

☑ Basic Training Algorithms

☑ Common Pitfalls & Training Considerations

☑ Searching Hyperparameters

Questions?