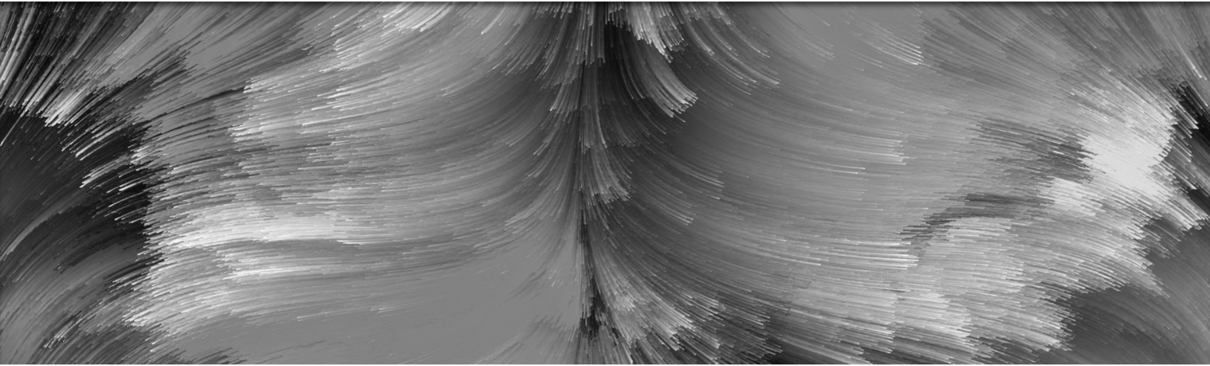
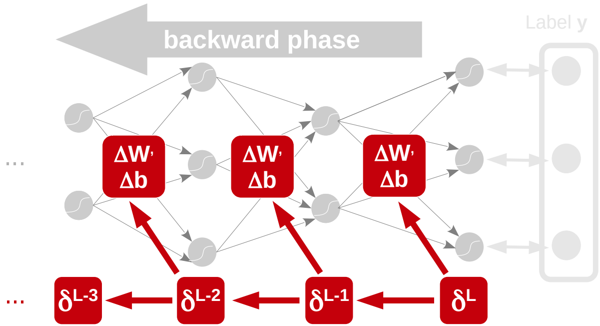


Künstliche Intelligenz (Sommersemester 2024)

Kapitel 08: Neuronale Netze: Tips & Tricks

Prof. Dr. Adrian Ulges

Backpropagation (Wiederholung)



Backpropagation-Formeln

$$\delta^L = \left(\mathbf{a}^L - \mathbf{t}\right) \odot f'(\mathbf{z}^L)$$

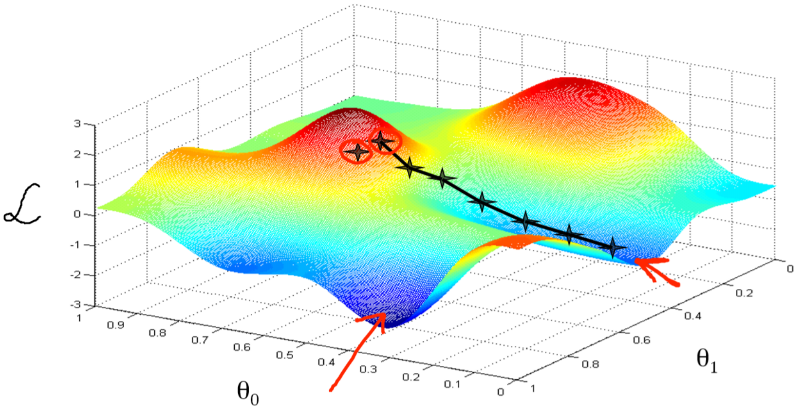
$$\delta^l = \left(W^{l+1} \cdot \delta^{l+1}\right) \odot f'(\mathbf{z}^l)$$

$$\Delta w_{ji}^l = -\lambda \cdot \delta_j^l \cdot a_i^{l-1}$$

$$\Delta b_j^l = -\lambda \cdot \delta_j^l$$

Rückpropagation

- Erreicht Backpropagation immer das globale Minimum der Loss-Funktion $\mathcal{L}$?
- Nein – weil die Optimierung per Gradientenabstieg erfolgt.



1990er: Enttäuschung

1. Gradientenabstieg ...

... ist langsam, nur lokal optimal und schwierig zu konfigurieren.

2. Lost in Hyperparameter Space

Neuronale Netze besitzen eine Vielzahl von Hyperparametern:

#Schichten/#Neuronen? Aktivierungsfunktionen? Topologie? Lernrate? Optimierer? Batch-Größe? Initialisierung? Normalisierung? Regularisierung? Loss-Funktion? ...

Neuronale Netze: Konzepte

Um neuronale Netze erfolgreich zu trainieren, müssen wir einige Effekte+Tricks verstehen:

Rectified Linear Units

Early Stopping

Learning Rate Schedules

Batch Normalization

Backpropagation Through Time

Dead Neurons

Dropout

Mini-Batching

Class Weighting

Saturated Neurons

Vanishing Gradients

Exploding Gradients

Optimizers

Cross-Entropy Loss

Residual Layers

Symmetry Breaking

Regularization

Layer Normalization

Convolutional Layers

Outline

1. Lernrate + Optimizer
2. Class Balancing + Saturierte Neuronen
3. Early Stopping + Regularisierung

Backpropagation: Mini-Batching Bild: [2]

- Backpropagation *(letzte Vorlesung)* aktualisiert die Gewichte nach jedem Trainingsbeispiel. Dies wird als stochastischer Gradientenabstieg (SGD) bezeichnet.
- In der Praxis wählen wir stattdessen je Iteration eine zufällige Menge *(sog. Mini-Batch)* von B Beispielen. Wir berechnen die Gewichtsaktualisierungen $\Delta w_1, ..., \Delta w_B$ pro Beispiel parallel und verwenden den Durchschnitt als Aktualisierung:

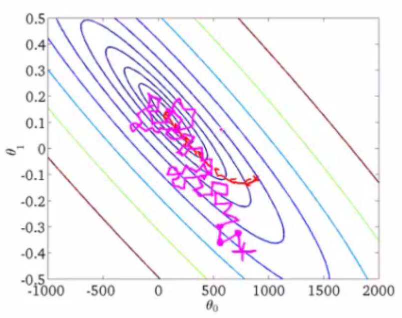
$$\Delta w := \frac{1}{B} \cdot \sum_{i=1}^{B} \Delta w_i$$

Was ist eine gute Batchgröße B?

Eine größere Batchgröße macht die Optimierung...

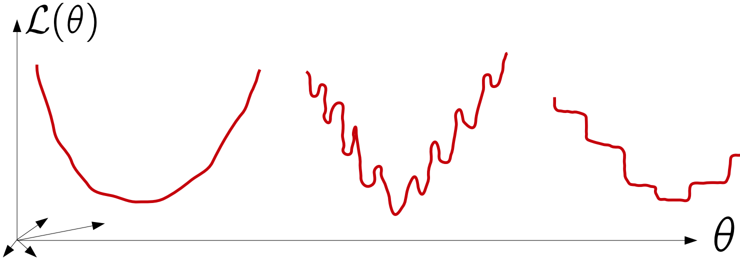
1. glatter *(→ größeres λ möglich)*
2. teurer *(mehr Δw_i's je Schritt zu berechnen)*.

Heuristik: Wähle das größte B, das in den (GPU-)Speicher passt. → beste Parallelisierung.



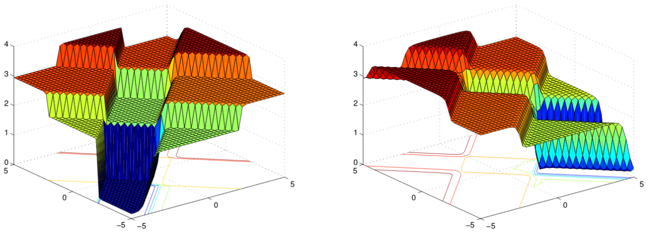
Kostenfunktionen in Backpropagation

Backpropagation ist Gradientenabstieg auf einer hochdimensionalen Verlustfläche $\mathcal{L}$.
Aber wie sieht $\mathcal{L}$ “aus”?



> *“Most practitioners believed that* **local minima** *were a common problem plaguing neural network optimization. Today, that does not appear to be the case. [...] Local minima are in fact rare compared to another kind of point with zero gradient: a* **saddle point**.*”*
>
> (Courville et al. [1])

Kostenfunktionen in Backpropagation: Beispiel



Die Wahl einer **guten Lernrate λ** ist schwierig!

- **λ zu klein**: Das Lernen ist langsam und "verhungert" auf Plateaus.
- **λ zu hoch**: Das Lernen springt wild hin und her und konvergiert in ungünstigen Bereichen des Parameterraums.

> *"The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate."*
>
> (Courville et al. [1])

Lernrate: Empfehlungen

Faustregel

Erkunde mit Faktoren von 3: Beginne mit einer Schätzung der Lernrate (z.B. λ_0=0.01) und trainiere ein wenig. Starte dann weitere Trainings mit $3\lambda_0, 9\lambda_0, \frac{1}{3}\lambda_0, \frac{1}{9}\lambda_0, ...$ und versuche eine Lernrate zu finden, die zu schnelleren Verbesserungen des Losses führt.

Learning Rate Schedules

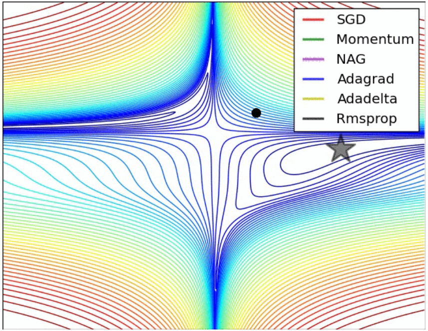
- Manchmal wird die Lernrate über den Verlauf des Trainings, d.h. mit der Trainingsiteration t=0, 1, 2, ..., variiert.
- **Beispiel**: Linearer Zeitplan mit zwei verschiedenen Lernraten $\lambda_{high}, \lambda_{low} \in \mathbb{R}^+$.

$$\lambda_t := (1 - \alpha_t) \cdot \lambda_{high} + \alpha_t \cdot \lambda_{low}$$

  wobei $\alpha_t = min(t/t_{max}, 1)$ (wir trainieren für t_{max} Iterationen).
- Oft wählt man eine etwas geringere Lernrate zu Beginn (sog. "burn-in phase") und zum Ende des Trainings (um ein "Überspringen" von Optima zu vermeiden).

Erweiterungen von Stochastischem Gradientenabstieg

- Es gibt viele SGD-Varianten, sogenannte Optimierer, die versuchen die Lernrate geschickter automatisiert zu wählen.
- Beispiele: Adam, AdaGrad, Nesterov Accelerated Gradient (NAG), ...
- Dieser tolle Blogbeitrag[1] erklärt die wichtigsten.



[1] http://ruder.io/optimizing-gradient-descent/index.html

Kernkonzept von Optimizern: Momentum Bild: [1]

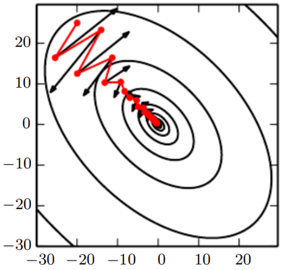
- Idee: **Glätten des Gradienten** über Trainingsiterationen.
- Es sei $\nabla \mathcal{L}^t$ der **Gradient** (d.h., das Gewichts-Update) in Iteration t.
- Wir definieren den laufenden Durchschnitt des Gradienten (mit $0 < \alpha < 1$):

$$\mathbf{v}^t = \alpha \cdot \mathbf{v}^{t-1} + \nabla \mathcal{L}^t$$

- Wir aktualisieren die Gewichte (und Biaswerte):

$$W := W - \lambda \cdot \mathbf{v}^t \; \left(\text{anstatt } W := W - \lambda \cdot \nabla \mathcal{L}^t \right)$$

- Mit Momentum ist die Richtung des Trainings stabilisiert, das Training 'rollt' über Plateaus hinweg.
- α ist die Stärke des Momentums.



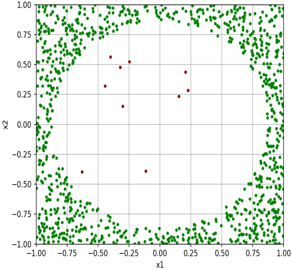
Outline

1. Lernrate + Optimizer
2. Class Balancing + Saturierte Neuronen
3. Early Stopping + Regularisierung

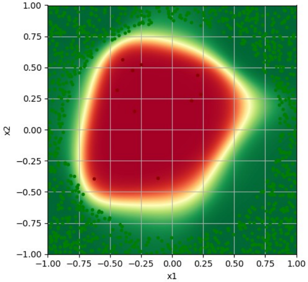
Unbalancierte Trainingsdaten

- Manchmal können wir viele Trainingsbeispiele aus einer Klasse sammeln, aber nur wenige aus einer anderen: Die Trainingsdaten sind unbalanciert.
- Beispiel: Betrugserkennung in Transaktionen.

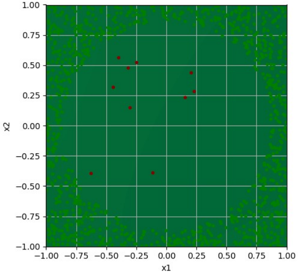
Neuronale Netze haben Schwierigkeiten mit ungleichgewichteten Datensätzen



Trainingsdaten
(unbalanciert)



Was das Netz
lernen **sollte**



Was das Netz
lernt

Unbalancierte Trainingsdaten: Lösung

Strategie 1: Subsampling

- Wir entfernen zufällig Trainingsbeispiele der überrepräsentierten Klasse.
- Daumenregel: Das Klassenverhältnis sollte mindestens 1/10 betragen.
- Nachteil: Wir verlieren Trainingsbeispiele. ☹

Strategie 2: Klassen-Gewichte

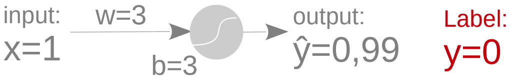
- Modifiziere den Loss, um Fehler der unterrepräsentierten Klasse stärker zu bestrafen als Fehler der überrepräsentierten Klasse.
- Beispiel: Quadrierter Fehler für Trainings-Inputs $\mathbf{x}_1, \ldots, \mathbf{x}_n$

$$\mathcal{L} = \sum_{i=1}^{n} w_i \cdot \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i)$$

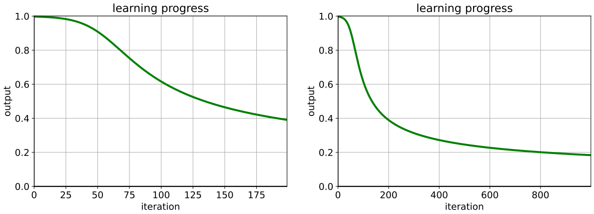
- Wir wählen w_i für die Trainingsbeispiele seltener Klassen höher (z.B. antiproportional zur Klassenhäufigkeit).

Saturierte Neuronen

Beispiel: Schlecht initialisiertes Neuron [3]



- Wir **trainieren das Neuron**, indem wir ihm wieder und wieder Eingabe x=1 und Label y=0 präsentieren.
- Wir plotten den **Lernfortschritt** des Neurons über die Iterationen:
  Wie schnell bewegt sich die Ausgabe $\hat{y}$ in Richtung der Soll-Ausgabe 0?



Gesättigte Neuronen

Warum ist das Lernen so langsam in Gang gekommen?

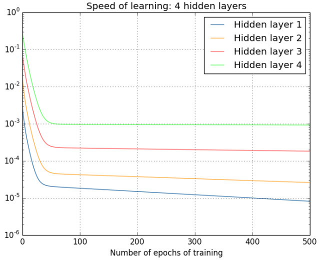
- Wir leiten das Gewichtsupdate Δw mit den Backpropagation-Formeln her...
  (wobei $\hat{y}$ die Ausgabe des Neurons und z seine eingehende Energie sind):

$$\Delta w = -\lambda \cdot \frac{\partial \mathcal{L}}{\partial w} = -\lambda \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w} = -\lambda \cdot (\hat{y} - y) \cdot g'(z) \cdot x.$$

- Das Problem: $g'(z)$ ist fast null, weil die Sigmoidfunktion g für Eingaben $z > 2$ oder $z < -2$ sehr flach ist.
- Wir nennen das Neuron saturiert. Saturierte Neuronen lernen schlecht. ☹
- Besser wäre, wenn das Neurone mit $z \approx 0$ initialisiert wäre. ☺

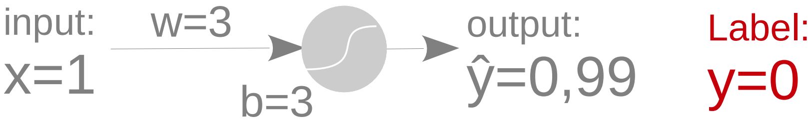
Vanishing Gradients Bild: [3]

- Mit tieferen Netzen wird dieses Problem immer schlimmer: Der Gradient verschwindet, je mehr Rückwärts-Schritte die Backpropagation macht.
- **Experiment**: Wir messen die **„Geschwindigkeit“ des Lernens** in den Schichten eines Beispiel-Netzs.



- Das Lernen in Schicht 1 ist **sehr langsam!** *(Beachten Sie die logarithmische Skala: Schicht 1 lernt 100× langsamer als Schicht 4).*

Zurück zu unserem saturierten Neuron...



- Wir wollen immer noch, dass das Neuron schneller trainiert!
- Idee: Unser Loss muss kleine Werte von g' kompensieren.
- Wir ersetzen unseren bisherigen Mean Squared Error (MSE) Loss...

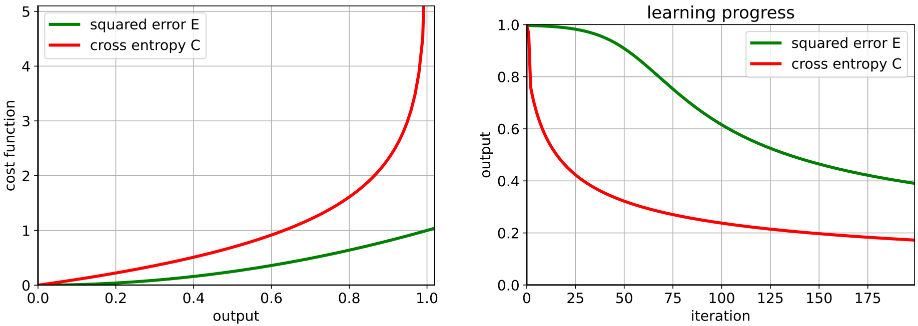
$$\mathcal{L} = (\hat{y} - y)^2$$

- ... mit dem sogenannten **Cross-Entropy**-Loss:

$$\begin{aligned} \mathcal{L}^{CE} &= -\Big(y \cdot log(\hat{y}) + (1-y) \cdot log(1-\hat{y})\Big) \\ &= -log(1-\hat{y}) \qquad // \text{ in unserem Fall} \end{aligned}$$

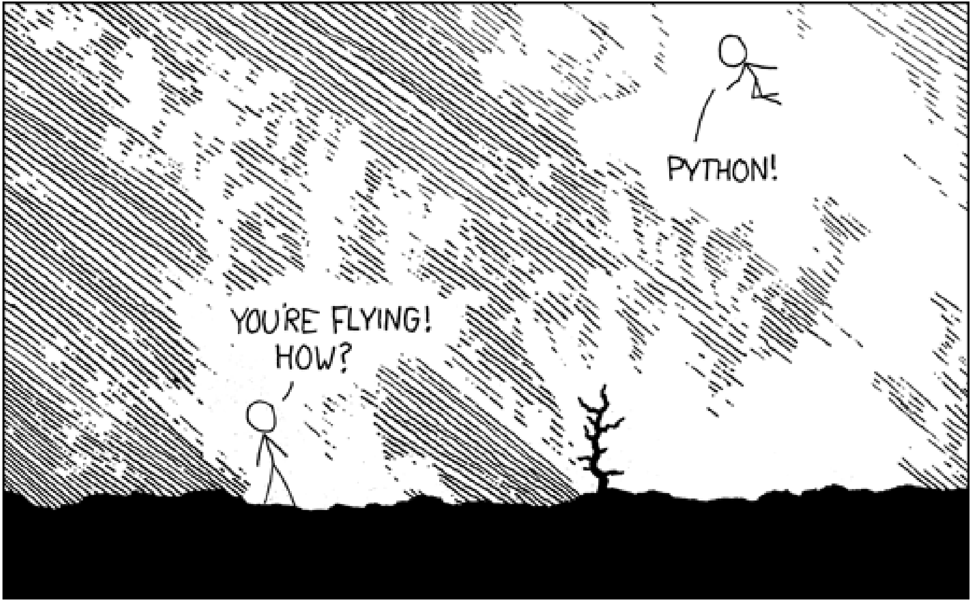
Cross-Entropy-Loss

- $\mathcal{L}^{CE}$ bestraft unser schlecht initialisiertes' Neuron **viel stärker** *(links)*.



- Mit der Cross-Entropy lernt unser Neuron viel schneller *(rechts)*!
- Viele Loss-Funktionen in der Praxis verwenden logarithmische 'Bestrafungs'-Terme *(z.B. Binary Cross-Entropy (BCE), Negatives Log-Likelihood (NLL), KL-Divergenz (KL-DIV))*.

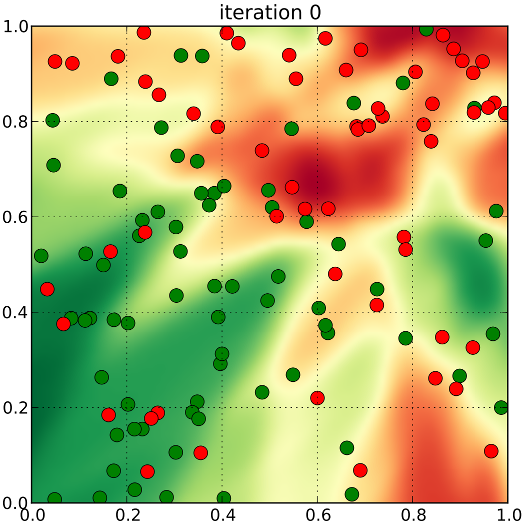
Notebook: Wie initialisieren wir unser Netz?



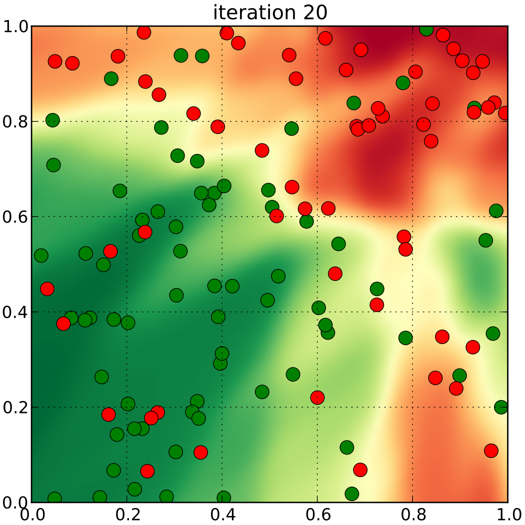
Outline

1. Lernrate + Optimizer
2. Class Balancing + Saturierte Neuronen
3. Early Stopping + Regularisierung

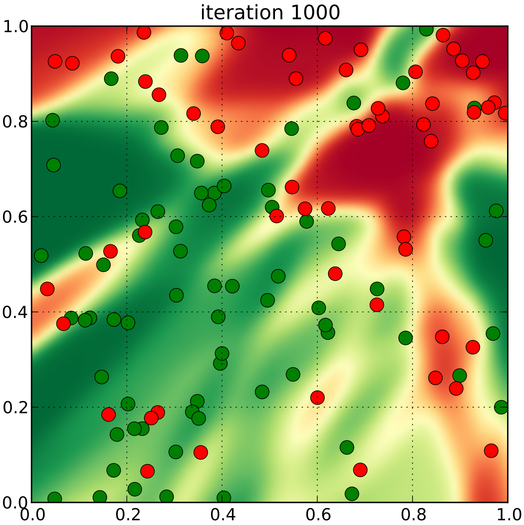
MLP: Beispieltraining



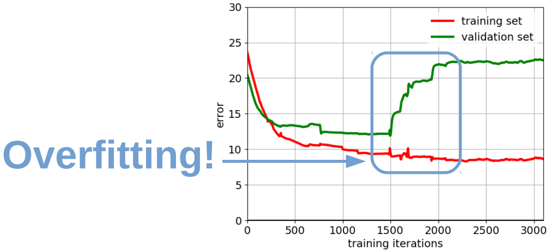
MLP: Beispieltraining



MLP: Beispieltraining



Wann soll man das Training beenden?



Übliche Praxis: "Early Stopping"

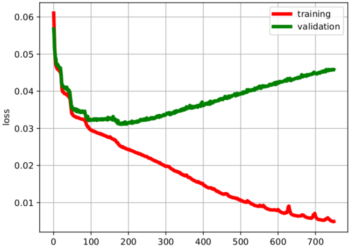
- Während des Trainings erfassen wir die Modellqualität auf einer Validierungsmenge.
- Als Endergebnis wählen wir das Modell aus dem Trainingsschritt direkt vor dem Zeitpunkt, an dem die Qualität auf dem Validierungssatz nicht mehr besser wurde.
- Eine sog. "Patience"-Schwelle legt fest, wie lange wir auf eine weitere Verbesserung warten bevor wir das Training abbrechen.

Overfitting

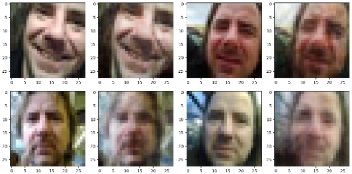
Oft haben neuronale Netze Probleme mit **Overfitting**: Netze, die **komplex** sind und **über viele Iterationen trainiert werden**, **spezialisieren sich zu stark** auf die Trainingsdaten.

Beispiel: Training von “PanitzNet”

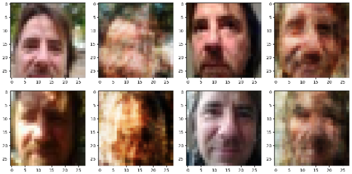
- **Oben**: Der Trainings-Loss (rot) verbessert sich weiter, der Validierungs-Loss (grün) steigt an.
- **Unten**: Wir sehen Paare von Eingabebildern und deren Rekonstruktionen.
- Auf den Trainingsdaten: gut ☺.
- Auf den Validierungsdaten: schlecht ☹.



reconstructions *(training)*



reconstructions *(validation)*



Bekämpfung von Overfitting: Zwei Strategien

Zwei gängige Strategien zur Bekämpfung von Overfitting:

1. **Regularisierung**: Vermeidung von extremen Gewichten.
2. **Dropout**: Zufällige Merkmale auslassen.

1. Regularisierung

- Wir definieren den Gewichtsvektor W, der alle Gewichte und Biasse des Netzs enthält *(oder nur die eines besonders komplexen Teils/Schicht).*
- $||W||_2^2$ ist die quadrierte (L2-)Norm von W:

$$||W||_2^2 \;=\; w_1^2 + w_2^2 + w_3^2 + \ldots$$

Vorgehensweise

- Ein "gutes" Netz enthält keine "extremen" Gewichtswerte.
  → die L2-Norm $||W||_2$ sollte klein sein.
- Wir **regularisieren** den Loss $\mathcal{L}$, indem wir einen Bestrafungsterm hinzufügen:

$$\mathcal{L}^{reg}(W) \;=\; \mathcal{L}(W) + \beta \cdot ||W||_2^2$$

1. Regularisierung

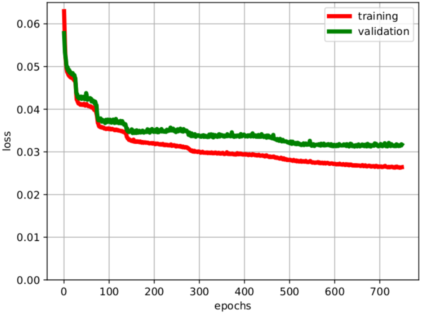
Der **Hyperparameter** β bestimmt die **Stärke der Regularisierung**:

- β klein: Es werden nur die Trainingsdaten optimiert. Starke Gefahr von Overfitting.
- β hoch: Modellgüte verschlechtert sich auf den Trainingsdaten, aber verbessert sich auf den Validierungsdaten *(weniger Overfitting)*.

Beispiel: Hohes β

reconstructions ***(training)***

reconstructions ***(validation)***

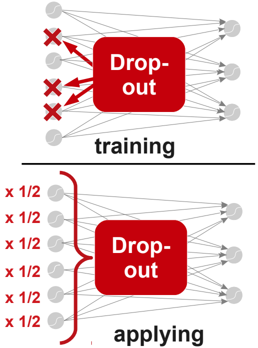


2. Dropout

- Das Netz sollte sich nicht zu stark auf ein bestimmtes Merkmal verlassen.
- Um das Modell robust zu machen, deaktivieren wir manche Neuronen zufällig.
- In PyTorch wird dies durch eine spezielle Schicht realisiert *(nn.Dropout())*.

Was macht Dropout?

- **Beim Training**: Dropout setzt zufällig $p\%$ (hier 50%) der Neuronen der vorherigen Schicht auf 0. p wird als Dropout-Rate bezeichnet.
- **Beim Anwenden des Modells**: Dropout multipliziert den Output aller Neuronen der vorherigen Schicht mit $(1-p)$, *damit die erwartete Eingabe für Neuronen der nächsten Schicht gleich wie beim Training ist*.



References I

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville.
Deep Learning.
Book in preparation for MIT Press (retrieved Nov 2016), 2016.

[2] A. Holehouse.
Stanford Machine Learning (Transcript of Course by Prof. Andrew Ng).
http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html (retrieved: Nov 2016).

[3] Michael Nielsen.
Neural Networks and Deep Learning.
Determination Press, 2015.