

Inhalt

1.	Introductory	5
	What is Machine Learning + Deep Learning.....	5
	AI:.....	5
	ML:.....	5
	DL:.....	6
	Neural Network Basics	6
	Artificial Neurons und Activation Funktion	6
	Gewichte.....	7
	Lernen in DNN	7
	CNN.....	7
	RNN.....	8
	End2End Learning.....	8
2.	Gradient Descent.....	9
	Supervised Learning	9
	Supervised Parametric Learning.....	9
	Predict	9
	Compare	10
	Learn (Weight adjustment)	10
	Basic Gradient Descent Algorithm.....	10
	Error/weight Relationship	11
	- Das einzige was wir ändern können, um error zu nullen, ist weight. Weil input, goal_pred und die Formel gleichbleiben müssen bzw. Änderung keinen Sinn macht.....	11
	Divergence.....	11
	Learning Rate.....	12
3.	Maths Refresher	13
	Linear Algebra.....	13
	Probability	14
	Differential Calculus	14
	Gradient Descent.....	14
4.	Feed Forward Neural Networks	15
	Gradient Descent.....	15
	Networks with multiple inputs	15
	Networks with multiple outputs	15
	Networks with multiple inputs and outputs	15
	Correlation.....	16

Learning Correlation.....	16
Creating Correlation	17
ACTIVATION FUNCTION.....	18
Backpropagation	20
Update Weights.....	20
Weights/Model.....	20
Bias	20
Loss Function.....	21
Loss & Activationfunction relationship	22
Grundregeln der Backpropagation.....	23
Optimierungsalgorithmen.....	25
Gradient Descent.....	25
Optimisierungstypen	26
5. CCNs & RNNs	28
CNN.....	28
Convolution	29
Convolutional Layers	29
Max Pooling.....	29
RNN.....	30
Network Structures	30
Backpropagation.....	31
Gated RNN.....	32
Backpropagation & Regularisation.....	36
Backpropagation	36
Generalisation	36
Generalisation of a supervised model.....	36
Data Partitioning	36
Bias und Varianzfehler.....	37
Generalisation Errors.....	37
Regularisation.....	38
Mini-Batch Learning	38
Weight Penalties.....	38
Data Augmentation	39
Training with Noise.....	39
Dropout	39
Early Stopping.....	40

Sequence To Sequence Learning.....	41
Natural Language Processing	41
Discrete Language Models	41
Continous Space Language Models.....	41
NLP with Deep Learning	42
Word Embedding.....	42
Neural Language Models.....	46
Sequence to Sequence Learning	48
Encoder.....	48
Encoder Vektor.....	48
Decoder	48
Training.....	48
Attention Mechanisms	49
Core Steps.....	49
Hierarchical Attention	49
Connectionist Temporal Classification	50
Pfad.....	50
Training.....	50
Decoding.....	50
Deep Reinforcement Learning.....	51
Introduction.....	51
Case: Interactive agent.....	51
Reinforcement Learning.....	51
Basics	51
Algorithmen.....	53
Deep Reinforcement Learning.....	55
Situation Sets.....	55
Allgemeiner Ansatz.....	55
Approximator Ansatz.....	55
Q-Network.....	55
Deep Q Learning	56
Deep Double Q-Learning	56
Evolving Learning.....	56
Next Generation Neural Networks.....	57
Spiking NN	57
Neural Turing Machines	57

Progressive Neural Networks	58
Residual Networks.....	58
Squeeze Nets	59
Bayesian Neural Networks	59

1. Introductory

What is Machine Learning + Deep Learning

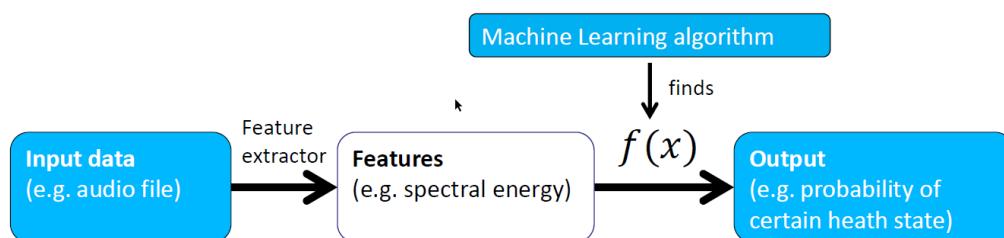
(AI(ML(DL)))

AI:

- Maschinen denken wie Menschen

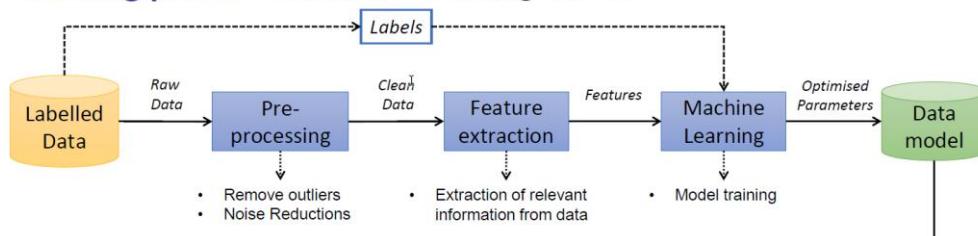
ML:

- Daten werden benutzt, um automatisch Tasks zu verbessern
- Daten und Antworten einspeisen, um Regeln zu erhalten
- Wird eher trainiert als programmiert, identifiziert strategische Struktur in Tasks, die Regelbildung erlaubt um diese auf neue Daten anzuwenden
- 3 Sachen für ML: Input Data, Expected Output, Maßstab ob der Algorithmus gut ist um current und expected Output zu vergleichen, anzupassen, zu lernen

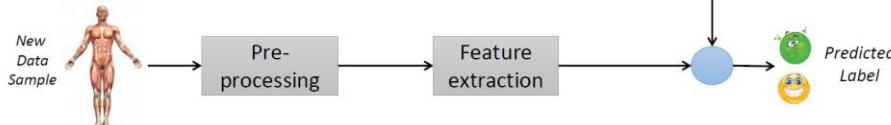


- Features ist Repräsentation der Daten, die dem ML-Algorithmus präsentiert werden, jedes Feature ist Single Information. Hunderttausende bilden Featurevektor. Alg identifiziert Patterns aus Vektorsammlung in iterativem Prozess.
- Ziel: Robuste, vorhersagende Funktion f zu erlernen, Mapping von Featurespace X zu Labelspace Y. $Y^* = f(x^*)$ ($*$ ist gleich Datensatz)

• Training phase – minimise training errors

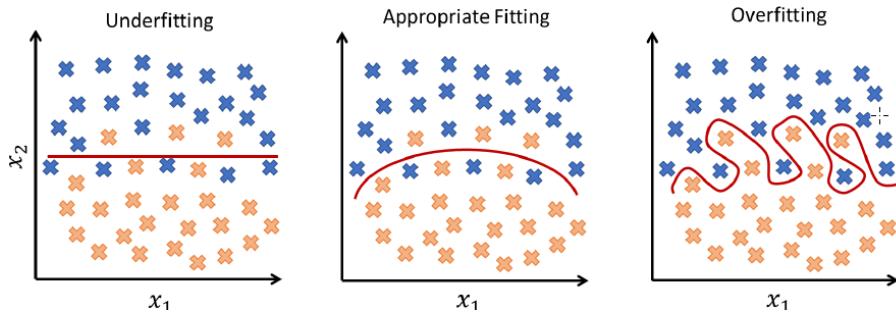


• Test phase – minimise generalisation errors



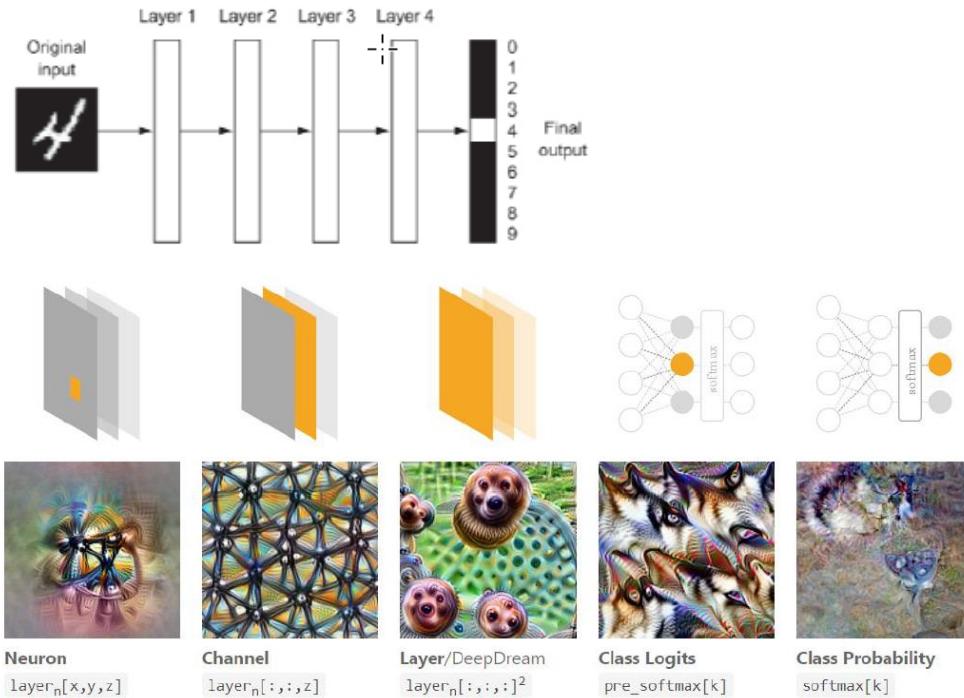
Generalisierungsfehler:

- Underfitting: Model ist zu einfach, Sensitivität fehlt
- Overfitting: Model ist zu komplex, versucht für alle Variationen Abgrenzungen



DL:

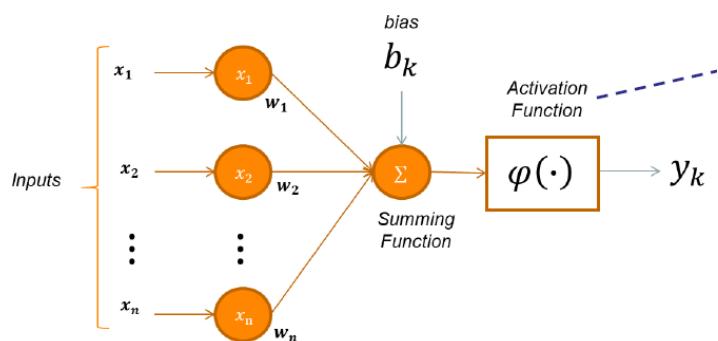
- Daten durch multilayered Netzwerk Neural Network wie Human Brain
- Spezielle Form von ML Algorithmen
- Lernen aufeinanderfolgender Darstellungsebenen („Deep“)



Neural Network Basics

Artificial Neurons und Activation Funktion

- Kombiniert verschiedene Inputs zu einzelnen Output
- Bilden Inputlayer, jedes Neuron nimmt einzelne Information auf
- Weight und Bias gehen in Activation Function, so wird bestimmt ob oder wie stark Neuron aktiviert wird

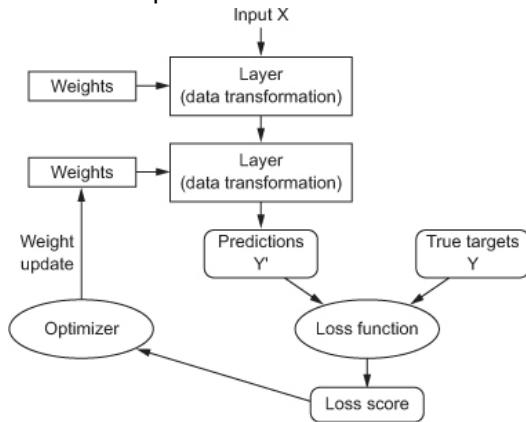


Gewichte

- Bestimmen Einflussgrad einzelner Neuronen auf Berechnung der Aktivierung
- Werden mithilfe von Gradient Descent gelernt
- Ein einzelnes Gewicht zu ändern hat auf alle anderen Einfluss
- Repräsentiert Wissen

Lernen in DNN

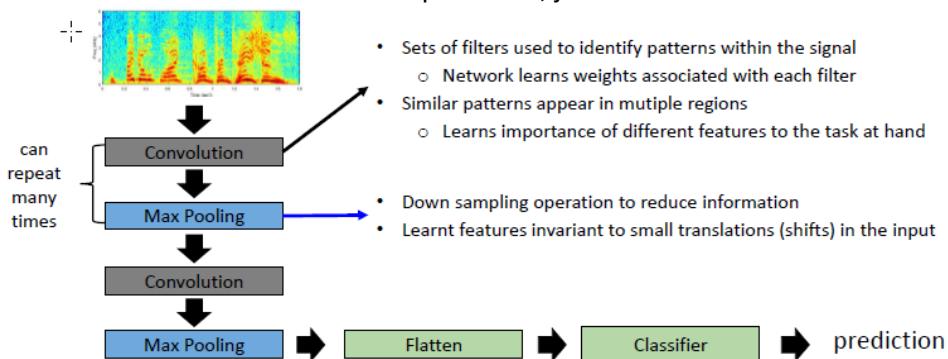
- Loss function wird benutzt, um Gewichtupdates zu kontrollieren: Wie krasse unterscheidet sich der Output vom Erwarteten

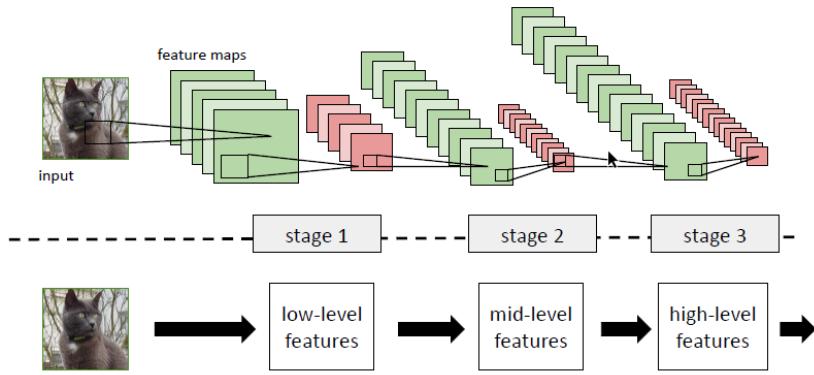


- DL gut bei riesigen Datasets (besonders bei Imageklassifizierung Natural Language und Speech)
- Universell einsetzbar: Netzwerk mit einzigen, großem verstecktem Layer ist für meiste Funktionen gut einsetzbar
- Im Gegensatz zum normalen ML braucht man keine Experten, weil Features selbst rausgefunden werden
-

CNN

- Besseres Featurelearning
- Erkennt mit Filtern Strukturen in Input Daten, je höher Filterebene desto abstrakter





RNN

- Im Gegensatz zu Feed Forward können hier Neuronen mit vorangegangenen oder gleichebigen Neuronen verbunden sein
- Hiddenoutput wird in Memory gestored und im nächsten Timestep wiederverwendet

End2End Learning

2. Gradient Descent

Supervised Learning

- Datensatz transferieren
- Input: Was wissen wir Output: Was wollen wir wissen
- 1. Welche Art von Daten soll Datensatz enthalten?
- 2. Datenerhebung entsprechend durchführen
- 3. Struktur von Funktion und Alg bestimmen
- 4. Lernalg auf Datensatz anwenden

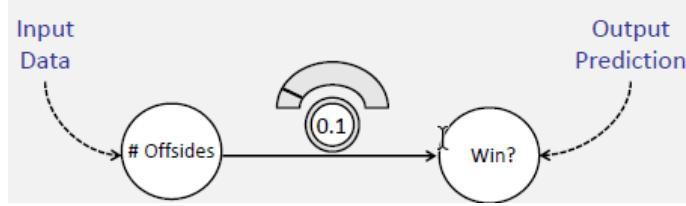
Supervised Parametric Learning

- Modell das Daten mit Parametern fester Größe zusammenfasst
- Zuerst Form für Funktion auswählen und dann Koeffizienten für die Funktion von Trainingsdaten lernen zB $b_0 + b_1x_1 + b_2x_2 = y$
- Analogie: Maschine mit fester Anzahl an Knöpfen, Knopfstellung sagt wie Daten processed werden sollen. Jeder Knopf repräsentiert die Sensitivität zu verschiedenen Arten an Input
- Processing transformiert input Daten zu output Vorhersage
- Learning dreht an den Knöpfen
- Keysteps sind: Predict, Compare with truth, Learn the pattern (dreh an Knöpfen)

Predict

Simple Prediction

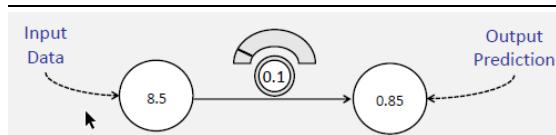
- Ein Datapoint input, einer Output
- Ein einzelner Knopf (Gewicht)



- 1. Netzwerk definieren 2. Input feeden 3. Value mit Weight multiplizieren 4. Output

```
weight = 0.1
def neural_network(input, weight):
    prediction = input * weight
    return prediction

number_of_offsides = [8.5, 9.5, 10, 9]
input = number_of_offsides[0]
pred = neural_network(input, weight)
print(pred)
```



- Prediction zu hoch -> weights lower Prediction zu low -> weights higher

Compare

- Wie richtig/falsch war die Prediction?

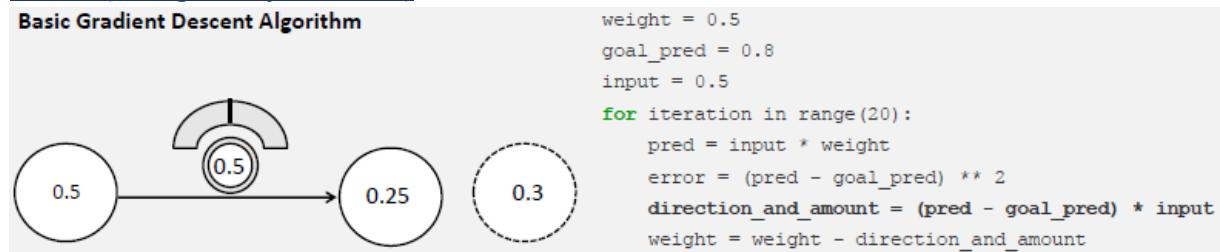


- Error^2 falls negativ \rightarrow Große Fehler werden größer, kleine kleiner

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

- Mean Square Error Equation:
- Weights so tunen, dass Error null ergibt (leichter als tunen um Target vorherzusagen)
- Wir wollen in riesigen Netzwerken den Durchschnitterror auf 0 bringen. Deshalb brauchen wir positive Errors, damit sich diese nicht mit den negativen ausnullen.

Learn (Weight adjustment)



- Direction und Amount: Wieviel muss ich das Gewicht ändern, um Error zu reduzieren
- Pure Error ist $\text{pred} - \text{goalpred}$, falls positiv ist Pred zu hoch, falls negativ zu niedrig
- Scaling: Weightchange ist proportional zu Input
- Negative Reversal: Sichergehen, dass Gewicht in richtige Richtung angepasst wird selbst wenn Input negativ
- Stopping: Wenn Input zero, wird nichts angepasst

Basic Gradient Descent Algorithm

- Goldene Methode für Neurales Lernen

```

pred = input * weight

error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = delta * input

weight = weight - weight_delta

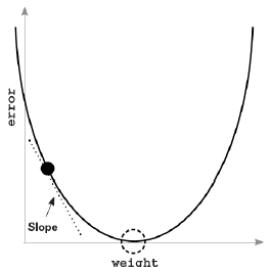
```

- Regelt jedes Gewicht in die korrekte Richtung mit dem korrekten Betrag, sodass Error auf null geht

Error/weight Relationship

- Für jedes input/pred Paar kann Relationship zwischen Error und Gewicht definiert werden, wenn man prediction und error Formeln kombiniert

```
error = ((input * weight) - goal_pred) ** 2
```



Wenn der schwarze Punkt der momentane Punkt ist, zeigt Slope immer auf Minimum der Funktion

- Das einzige was wir ändern können, um error zu nullen, ist weight. Weil input, goal_pred und die Formel gleichbleiben müssen bzw. Änderung keinen Sinn macht

Zu lernen bedeutet, das weight zu beeinflussen, um error zu nullen. Ziel: Kenne die Richtung und den Wert wie sich der Error verhält, wenn du das Gewicht anpasst.

Mit Derivativen kann man in jeder Formel zwei Variablen auswählen und schauen wie sie zueinanderstehen. Das können wir nutzen, um das Weight so anzupassen, damit der Error auf 0 geht (das ist unser weighted_delta).

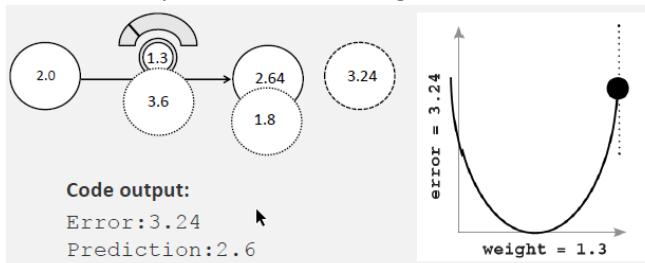
- Für jeden Punkt auf dem Graphen sagt es uns, um wieviel der Error sich verändert, wenn wir weight anpassen. Ist immer in die gegenteilige Richtung von minimum gerichtet. Deshalb moved man immer in die entgegengesetzte Richtung, wenn man „weight = weight – weight-delta“ berechnet

Beim Lernen wird solange iteriert bis Error 0, immer wieder weight anpassen bis schwarzer Punkt den Graphiefpunkt erreicht

Divergence

```
weight = weight - (input * (pred - goal_pred))
```

- Wenn der Input groß ist wird das Weightupdate automatisch auch groß, selbst wenn der Error klein ist. Dies kann dazu führen, dass das Update overshootet/overcorrectet und sich vom Wunschpunkt entfernt. Bei großem Error umso mehr



- Deshalb führt man zusätzliche variable „alpha“ ein, um das Gewichtupdate zu skalieren. Üblicherweise mit irgendwas zwischen 0 und 1.

```
weight = weight - (alpha*derivative)
```

Learning Rate

- Wie bestimmt man Alpha? Mit Erfahrung!
- Wenn Divergenz eintritt ist Alpha zu hoch, wenn man zu langsam lernt ist es zu niedrig

Final Basic Gradient Descent for Neural Learning:

```
pred = input * weight  
error = (pred - goal_pred) ** 2  
derivative = input * (pred - goal_pred)  
weight = weight - (alpha * derivative)
```

Derivative sagt Richtung und Menge für Weightadjustment vorher
Alpha hilft, Divergenzeffekte zu minimieren, wenn Input groß

3. Maths Refresher

Linear Algebra

VEKTOR NORM

- ℓ^1 norm example

$$v = \begin{bmatrix} 1 \\ -4 \\ 5 \end{bmatrix}, \|v\|_1 = |1| + |-4| + |5| = 10$$

- ℓ^2 norm example

$$v = \begin{bmatrix} 1 \\ -4 \\ 5 \end{bmatrix}, \|v\|_2 = \sqrt{|1|^2 + |-4|^2 + |5|^2} = \sqrt{42}$$

DOT PRODUCT

$$v_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, v_2 = \begin{bmatrix} 3 \\ 5 \\ -1 \end{bmatrix}$$

$$v_1 \cdot v_2 = v_1^T v_2 = 1 \times 3 + 2 \times 5 - 3 \times 1 = 10$$

- Winkel zwischen 2 Vektoren, wie ähnlich sind sich 2 Vektoren?

MATRIX MULTIPLIKATION

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}, C = A \times B$$

$$C_{11} = [1 \quad 2] \begin{bmatrix} 5 \\ 7 \end{bmatrix}, 1 \times 5 + 2 \times 7 = 19, C_{12} = [1 \quad 2] \begin{bmatrix} 6 \\ 8 \end{bmatrix}, 1 \times 6 + 2 \times 8 = 22$$

$$C_{21} = [3 \quad 4] \begin{bmatrix} 5 \\ 7 \end{bmatrix}, 3 \times 5 + 4 \times 7 = 43, C_{22} = [3 \quad 4] \begin{bmatrix} 6 \\ 8 \end{bmatrix}, 3 \times 6 + 4 \times 8 = 50$$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

MATRIX TRANSPOSE

- Die Spalten als Reihen Transponieren: Wichtig, weil $[a^*b] = a^T b$

DETERMINANT

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$= a_{11}(a_{22}a_{33} - a_{32}a_{23}) - a_{21}(a_{12}a_{33} - a_{32}a_{13}) + a_{31}(a_{12}a_{23} - a_{22}a_{13})$$

MATRIX INVERSE A^{-1}

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^{-1} = \frac{1}{a_{11} \times a_{22} - a_{21} \times a_{12}} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

$$\begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}^{-1} = \frac{1}{4 \times 6 - 2 \times 7} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix} = \frac{1}{10} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix} = \begin{bmatrix} 0.6 & -0.7 \\ -0.2 & 0.4 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}^{-1} = \frac{1}{3 \times 8 - 6 \times 4} \begin{bmatrix} 8 & -4 \\ -6 & 3 \end{bmatrix} \rightarrow \text{inverse does not exist}$$

Probability

- ML hat immer Unsicherheiten
- Wahrscheinlichkeitsgesetze bestimmen, wie ML Algorithmus schließen sollte

Differential Calculus

Gradient Descent

4. Feed Forward Neural Networks

Gradient Descent

Networks with multiple inputs

- Jeder Input hat eigenes Gewicht
- Output ist Summe aus Inputs * Weights

Alg:

1. Output = Input[0] * Weight[0] + Input[1] * Weight[1] ...
2. Error = (Output – Tatsächlich)²
3. Delta = Output – Tatsächlich
4. Weight_delta[i] = input[i] * Delta
5. New_Weight[i] = Old_Weight[i] – alpha * Weight_delta[i]

delta: Hey, inputs. Next time, predict a little higher.

Single weight:

- **Stopping:** if my input was 0, then my weight wouldn't have mattered, and I wouldn't change a thing
- **Negative Reversal:** if my input was negative, then I'd want to decrease my weight instead of increase it
- **Scaling:** my current input is positive and quite large, so my personal prediction was important I'm going to move my weight up a lot to compensate

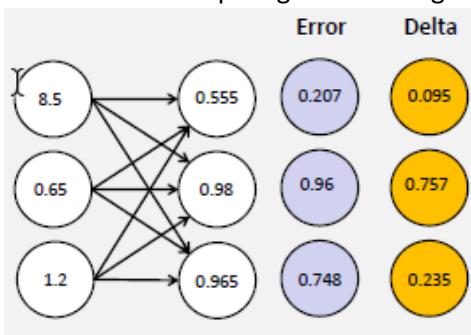
Dann wird iteriert solange bis Error nahe 0 (selber Input)

Networks with multiple outputs

- Netzwerk verhält sich wie 3 unabhängige Komponenten mit selbem Input
- Alle Variablen individuell voneinander berechnen, außer eben Input

Networks with multiple inputs and outputs

- Straightforward, alle Inputs werden in allen Outputs verarbeitet
- Bei 3 In und 3 Outputs gibt es 9 Weights

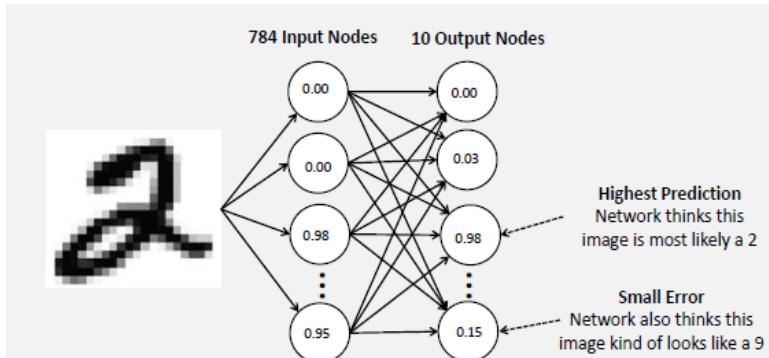


- Aus jedem Error und Delta werden die neuen Weights berechnet

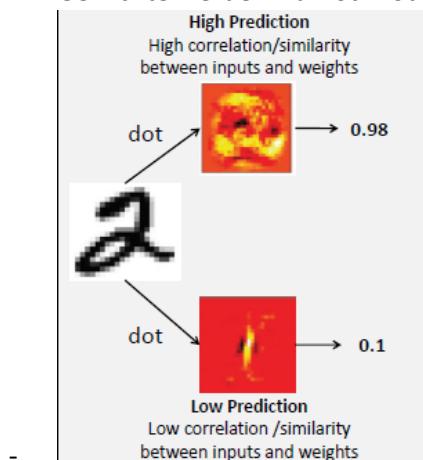
Correlation

Learning Correlation

- Was lernen die Gewichte im Ganzen?



- Wenn ein Gewicht hoch ist, denkt das Model, dass es viel Korrelation zwischen diesem Input und der Prediction gibt
- Gewichte werden via Dot Products gefunden und Dot Products kodieren Ähnlichkeit



Wie lernt man mit kompletten Datensätzen?

- Alle Daten iterieren und dann erst neuen Trainingszyklus starten

```
import numpy as np
weights = np.array([0.5, 0.48, -0.7])
alpha = 0.1
streetlights = np.array([
    [1, 0, 1],
    [0, 1, 1],
    [0, 0, 1],
    [1, 1, 1],
    [0, 1, 1],
    [1, 0, 1]
])
walk_vs_stop = np.array([0, 1, 0, 1, 1, 0])

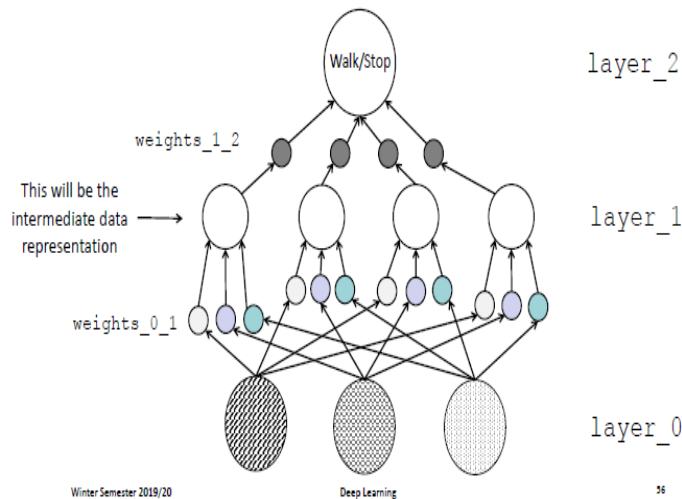
for iteration in range(40):
    error_for_all_lights = 0
    for row_index in range(len(walk_vs_stop)):
        input = streetlights[row_index]
        goal_prediction = walk_vs_stop[row_index]
        prediction = input.dot(weights)
        error = (goal_prediction - prediction) ** 2
        error_for_all_lights += error
        delta = prediction - goal_prediction
        weights = weights - (alpha * (input * delta))
        print("Prediction:" + str(prediction))
    print("Error:" + str(error_for_all_lights) + "\n")
```

- Manche Inputs bekommen mehr upward und manche mehr downwards pressure, das heißt nach einigen Iterationen wir sich ein Input als wertvoller als der andere herausstellen

- Jede Node versucht individuell den Output zu predicten, einziger Share ist der gemeinsame Error
- Lernen belohnt Korrelation mit größeren Weights, bestraft disccorr mit niedrigen

Creating Correlation

- Was wenn es keine Korrelation zwischen In- und Output gibt? Error wird nie 0 erreichen!
- Da ein Netzwerk Korrelation zwischen In- und Outputlayern sucht, nutzen wir einfach mehr Netzwerke bzw. mehr Layer: Layer 1 hat limitierte Korrelation zu Output, Hidden Layer nutzt diese um den Output zu predicten „Intermediate Datasets“



Weights der oberen Hälfte werden wie bisher geupdatet, Rest neu:

- Wenn man's so wie bisher (lineare Transformation) macht bringt das Hidden Layer nichts, weil mehrere Multiplikationen prinzipiell auch in einer gemacht werden könnten
- In linearem Netz haben Hidden keine eigene Korrelation, weil sie mit den Inputnodes korrelieren. Idealerweise sollen sie manchmal und manchmal nicht mit Input korrelieren („Conditional Correlation“)
- Hierzu braucht man nicht-lineares:

Beim Deep Learning geht's darum, Zwischenlayer zu kreieren. Jede Node darin repräsentiert das Dasein oder Nichtdasein einer anderen Inputkonfiguration

→ Kein individueller Input muss direkt mit Target korrelieren

Es geht also darum, eine Konfiguration von Inputs zu finden, die den Output kreieren, nicht einzelne Inputs sollen vom Output abhängig sein

ACTIVATION FUNCTION

- Benutzt den gewichteten Inputwert um das Level der Outputactivation zu bestimmen
- Wird auf Neuronen im Layer angewandt

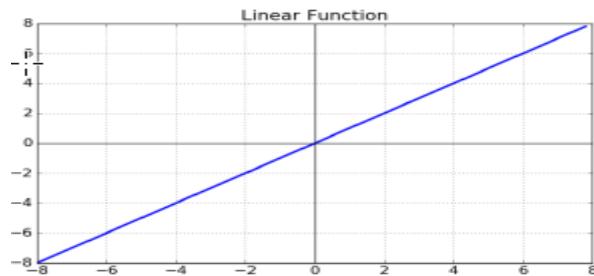
Grundeigenschaften

- Funktion hat unendlichen Wirkungsbereich
- Ändert nie die Richtung
- Sind nichtlinear
- Effizient zu berechnen

Linear

$$f(x) = ax$$
$$f'(x) = a$$

Range: $-\infty$ to ∞



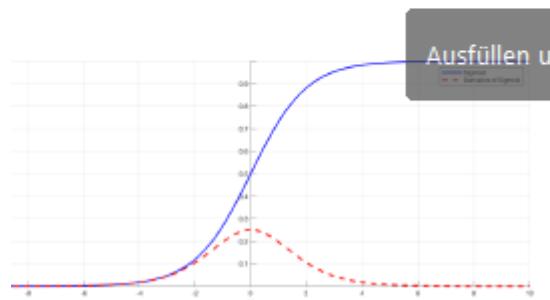
Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Range: 0 to 1

Komplexer und wird nicht allzu groß, aber bei großem x wird der derivative fast 0

- Vanishing Gradient möglich



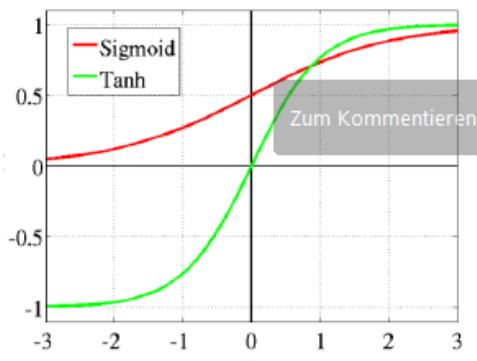
Hyperbolic Tangent

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\tanh'(x) = 1 - \tanh(x)^2$$

Range: -1 to 1

- Komplexere Patterns möglich
- Nicht allzu große Values
- Negativer Input gibt negativen Output
- 0 gibt 0
- Vanishing Gradient möglich



Zum Kommentieren

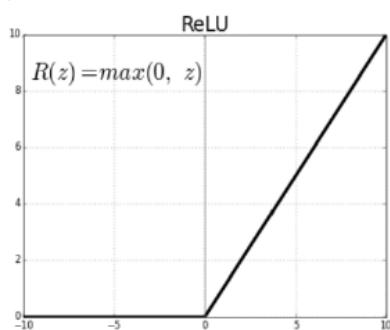
Rectified Linear Unit (ReLU)

$$ReLU(x) = \max(0, x)$$

$$ReLU'(x) = \begin{cases} 0, & \text{for } x < 0 \\ 1, & \text{for } x \geq 0 \end{cases}$$

Range: 0 to ∞

- Komplexe Patterns
- Kann sehr großen Output geben
- Leichte Ableitung
- Keine Negativen: Manche Strukturen koennten verpasst werden
- Dying RELU: gradient wird null und Gewicht wird nicht geupdatet



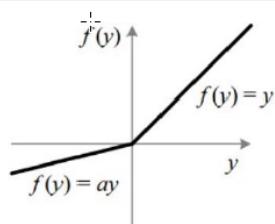
Leaky Rectified Linear Uni (LReLU)

$$LReLU(x) = \begin{cases} ax & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

$$LReLU'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Range: $-\infty$ to ∞

- Komplex, großer Output
- Leichte Ableitung
- Soll RELU Probleme lösen



Softmax

$$S(\mathbf{x}): \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix} \rightarrow \begin{bmatrix} s_1 \\ s_2 \\ \dots \\ s_N \end{bmatrix}$$

$$s(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

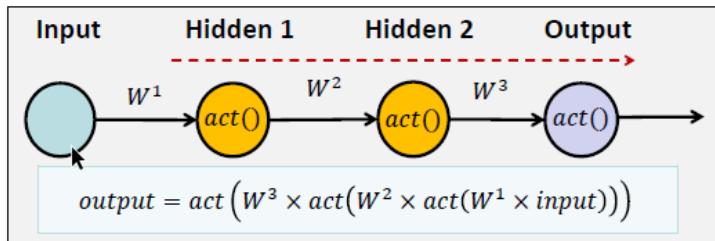
$$\frac{\partial s_i}{\partial x_j} = \begin{cases} s_i(1 - s_j), & \text{for } i = j \\ -s_i s_j, & \text{for } i \neq j \end{cases}$$

Range: 0 to 1

- Modelliert Wahrscheinlichkeitsverteilung
- Values zwischen 0 und 1, Summe gibt 1
- Typischerweise nur in Outputlayer benutzt

Backpropagation

- Bei Feed Forward geht's von links nach rechts



- Backpropagation ist Tool für mehrere Layer, Gewichte werden berechnet mit Respekt zu jedem Weight

Update Weights

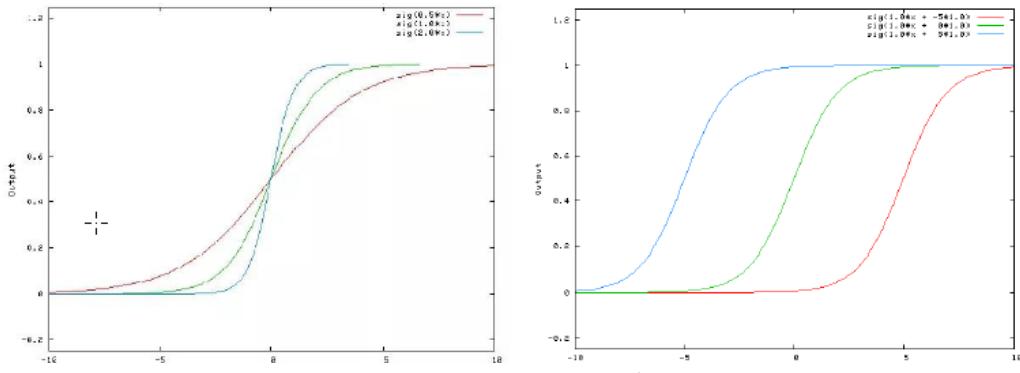
Weights/Model

Bias

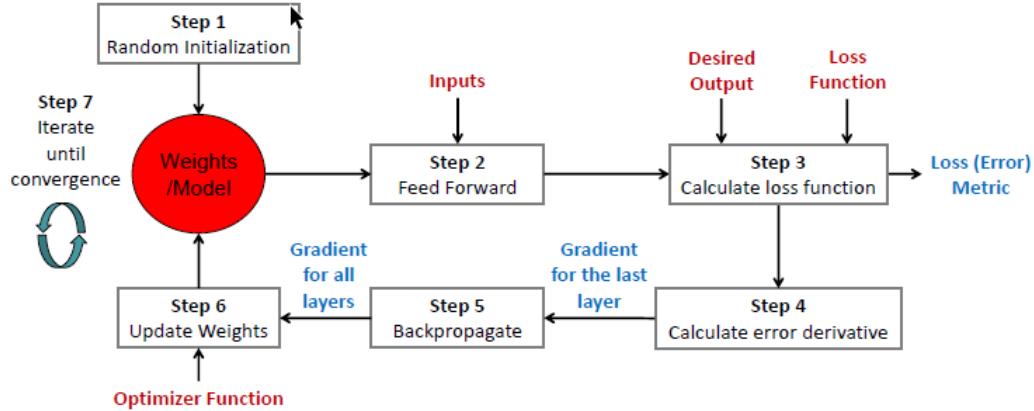
- Konstanter Vektor, das zu jedem Produkt Input*Weights hinzugefügt wird

```
output = activation_function(dot_product(weights, inputs) + bias)
```

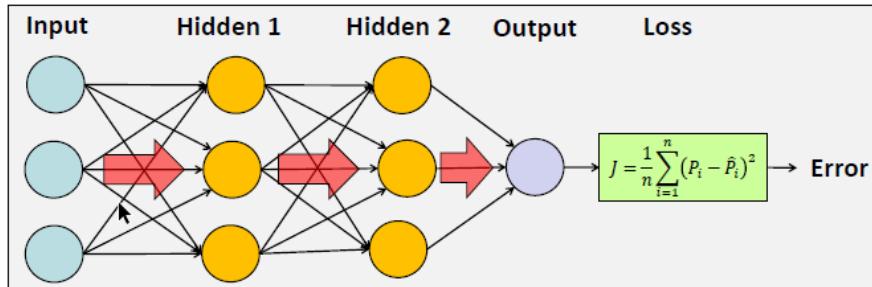
- Activationfunktion kann nach link und rechts angepasst werden, zur besseren Anpassung an die Daten
- Starten gewöhnlich mit 0 und werden vom Algorithmus angepasst, sind nicht abhängig vom Output früherer Layer



- Gewichtete beeinflussen Steigung(links), Bias den links/rechts Shift (bei Act. Function)



Loss Function



- Misst Differenz zwischen Vorhersage \hat{y} und was es vorhersagen hätte sollen
- Mit dieser Info kann das Model geupdated werden
- Loss ist Prediction error
- Loss Function berechnet den Loss
- Loss wird benutzt, um Gradient zu berechnen
- Gradient wird benutzt, um Weights upzudaten

Mean Squared Error (MSE)

$$\mathcal{L}(\hat{y}, y) = \sum_i (\hat{y}_i - y_i)^2$$

Üblicherweise für Regression genutzt

Cross Entropy

Für Klassifizierungen, das Model sagt die Wahrscheinlichkeit voraus, das eine Beobachtung ein bestimmtes Label hat

$$x \rightarrow \{p_c\}_c \text{ with } \sum_c p_c = 1$$

- Cross entropy measure the distance between two probabilities

$$\mathcal{L}(\tilde{p}, p) = \sum_c p_c \log(\tilde{p}_c)$$

nr 2019/20

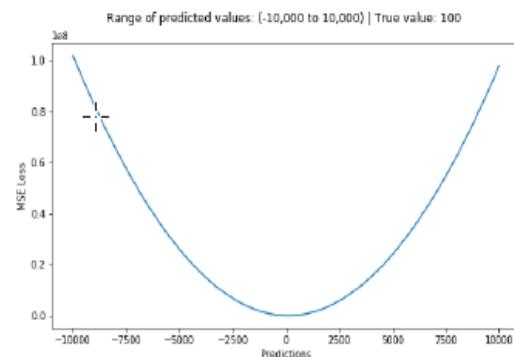
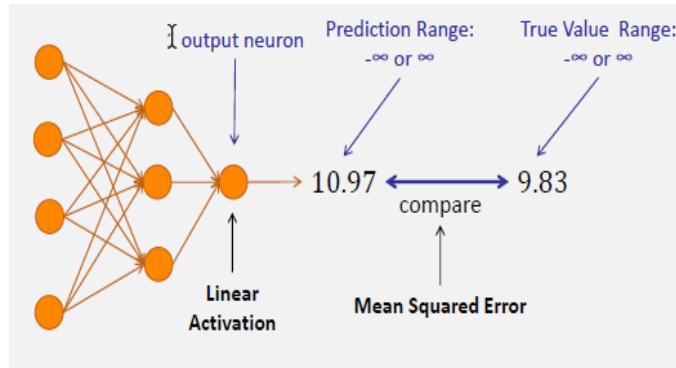
Loss & Activationfunction relationship

Je nach Output sollte man eine andere Costfunction wählen

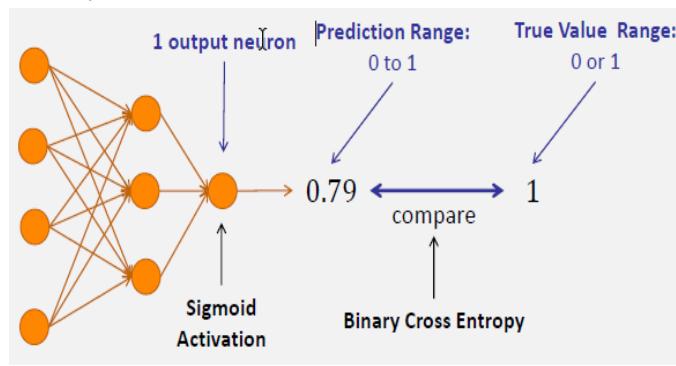
Problem Type	Output Type	Final Activation Function	Loss function
Regression	Numerical	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple label, multiple classes	Sigmoid	Binary Cross Entropy

- Was versucht man zu lösen/vorherzusagen?
- Binary: Data ist (k)eine Klasse?

Regression



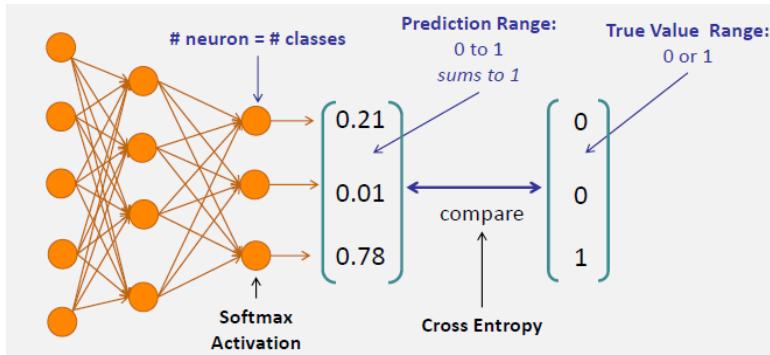
Binary Outcome



- Sigmoid führt zu Wert zwischen 0 und 1, aus dem man Klassenzuordnung schließen kann
- Binary Cross Entropy:
Model sagt Verteilung $\{p, 1-p\}$ vorraus
Wir vergleichen dies mit echter Verteilung $\{y, 1-y\}$

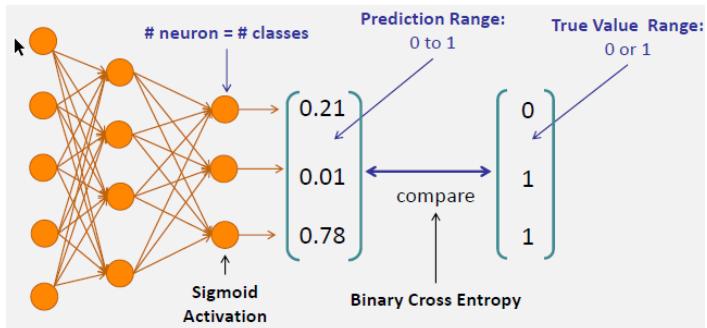
$$BCE = -(y \log(\hat{y}) + (1-y) \log(1-\hat{y}))$$

Single Label – Multiple Classes



- Softmax gibt Werte zwischen 0 und 1 die addiert 1 ergeben, Wahrscheinlichkeitsverteilung
 - Cross Entropy für Differenz
- $$CE = - \sum_{i=1}^M y_i \log(\tilde{y}_i)$$

Multiple Layers – Multiple Classes



- Das letzte Layer hat ein Neuron für jede Klasse, welches Wert zw. 0 und 1 zurückgibt
- Mit BCE vergleichen

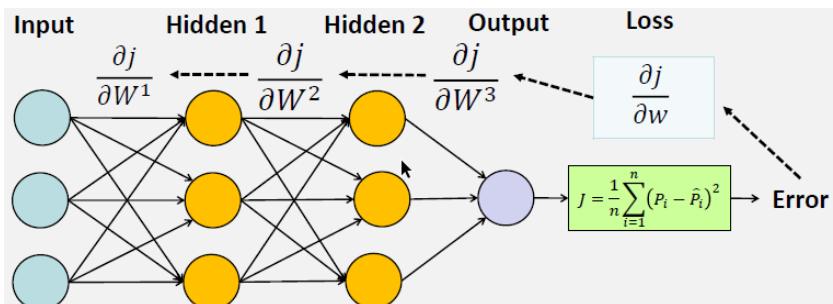
$$BCE = -(y \log(\tilde{y}) + (1 - y) \log(1 - \tilde{y}))$$

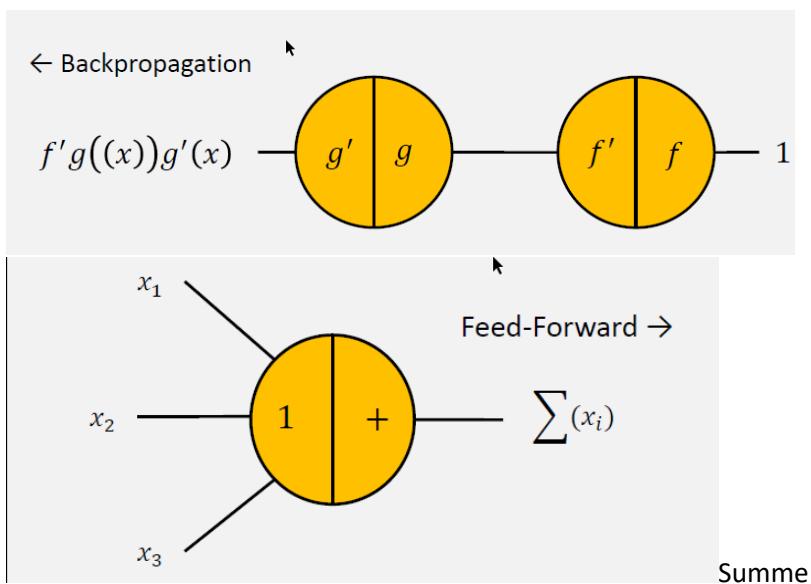
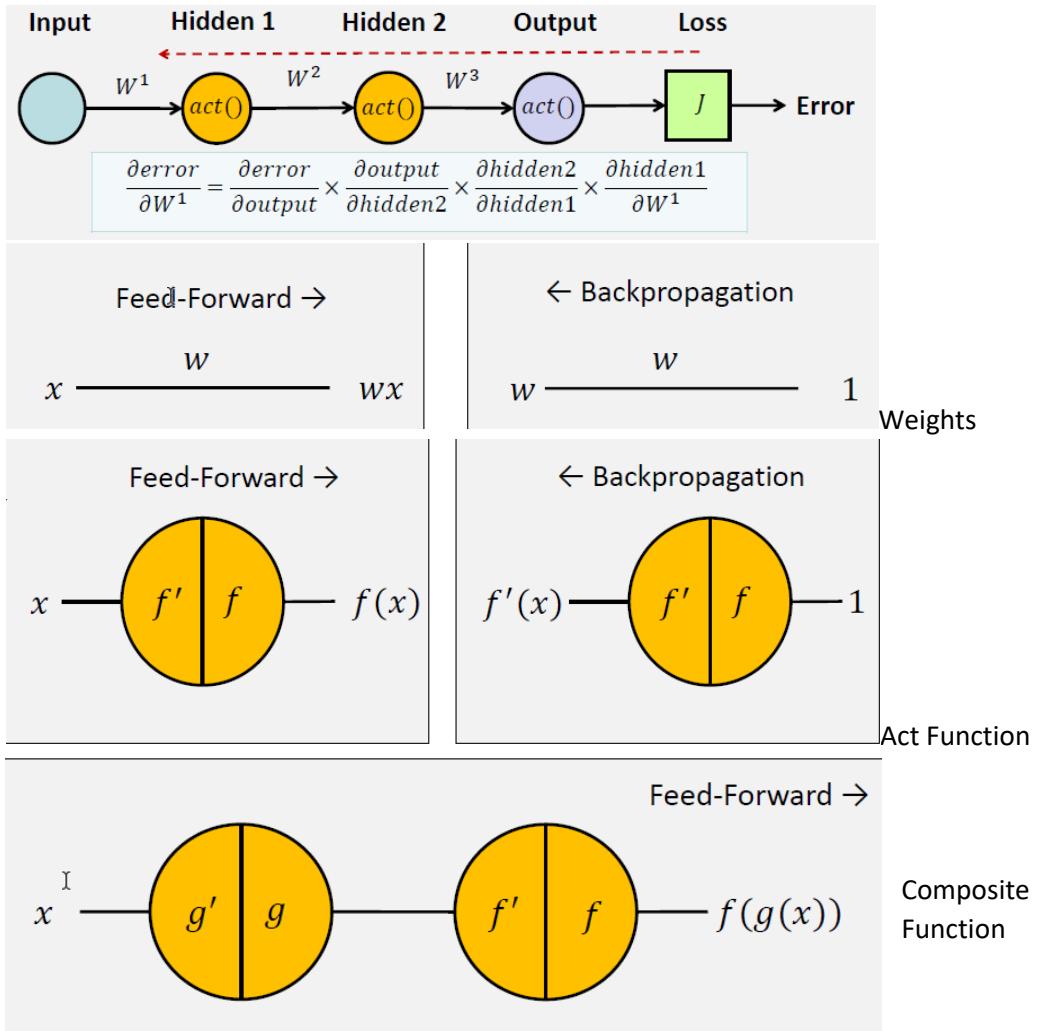
Grundregeln der Backpropagation

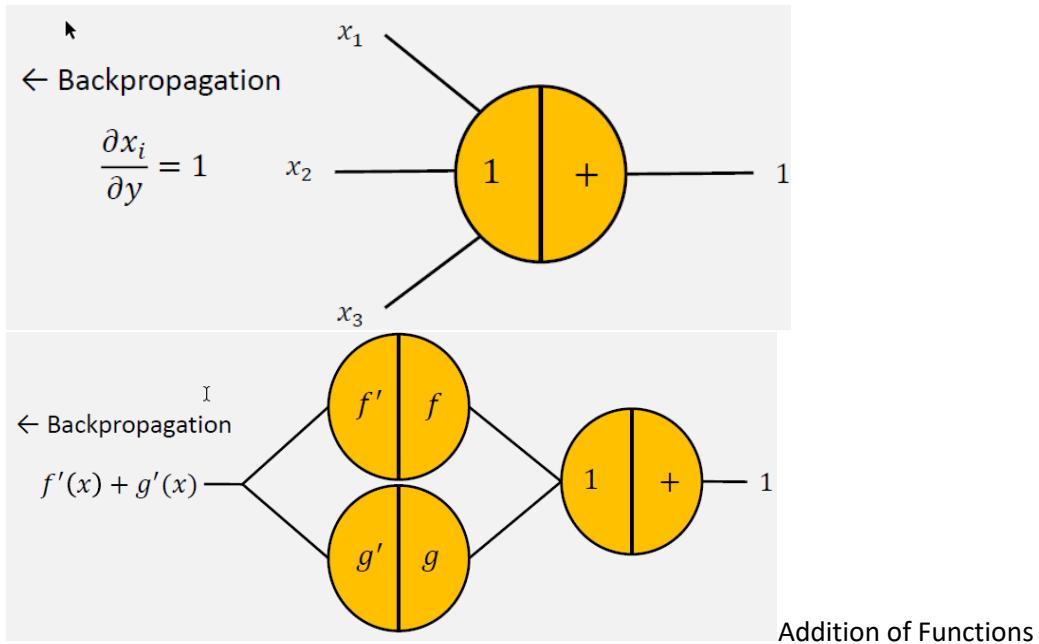
- Gewichte Updaten um Lossfunction zu minimieren
- Gradient der Lossfunction wird genommen, hierbei werden alle vorangegangenen Gewichte respektiert

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \alpha \nabla \mathcal{L}(\mathbf{W}^{(t)}; \mathbf{x}, \mathbf{y})$$

- Von hinten nach vorne werden Gewichte geupdated
- Gradient der Lossfunction wird berechnet







Optimierungsalgorithmen

Gradient Descent

- Gradient gibt Richtung des Increase an, deshalb verwendet man gegenteilige Richtung um das Minimum zu finden

Stochastic Gradient Descent

- Prediction und Weight für jedes Trainingsbeispiel selber geupdatet

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J(\theta^{(t)}; x^{(i)}; y^{(i)})$$

- Vorteile: Schneller als Batch, Gewichte leicht upgradebar wenn neue Beispiele benötigt
- Nachteile: Beim ständigen Update schwankt die Kostfunktion stark

Full Gradient Descent

- Berechnet durchschnittl. Weight-delta über komplettes Datenset, Updatet nur sobald kompletter Schnitt berechnet wurde

Batch Gradient Descent

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla C(\theta^{(t)}; x, y)$$

- Batchsize zwischen 8 und 256 aussuchen um weight_delta zu berechnen
- Respektiert beim Gradient berechnen das ganze Trainingsset
- Leicht zu verstehen
- Redundante Computation
- Langsam

MiniBatch

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J(\theta^{(t)}; x^{(i:i+n)}; y^{(i:i+n)})$$

- Minibatch für alle n Trainingsbeispiele
- Stabiler, schneller, gute Approximation
- Aber gesamter Loss nicht aufaddiert

Optimisierungstypen

Momentum

- Beschleunigt Annäherung des GD
- Nimmt jetzigen GD und den vorherigen Steps
- Hilft bei Kostfunktion mit Schlucht

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta; x, y)$$

$$\theta = \theta - v_t$$

Nesterov Accelerated Gradient

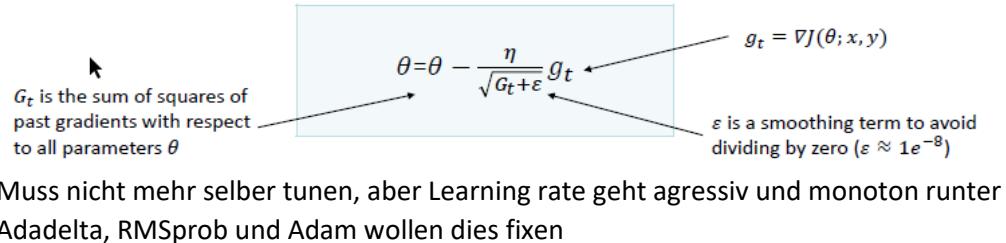
- Momentum nicht immer gut, manchmal muss man langsam machen
- Man nimmt den künftigen Step zum Kalkulieren

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta - \gamma v_{t-1}; x, y)$$

$$\theta = \theta - v_t$$

Adagrad – Adaptive Gradient Algorithm

- Teuer
- Learning rate passt sich an



Adadelta

- Grenzt das Miteinbeziehen vorangegangener Gradienten ein
- Die Lernrate wird dynamisch als Verhältnis des laufenden Durchschnitts der vorherigen Zeitschritte zum aktuellen Gradienten festgelegt

$$\begin{aligned} \theta_{t+1} &= \theta_t + \Delta\theta_t \\ \Delta\theta_t &= -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g_t]} g_t \end{aligned}$$

RMSProb

- Lernrate mit moving Average des squared Gradient anpassen

$$\begin{aligned} E[g^2]_t &= \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned}$$

Annotations:

- γ typically set to 0.9
- η typically set to 0.001

Adam

- Lernrate wird aus Schätzungen der ersten und zweiten Momente des Gradienten berechnet
- Effizient und wenig Memory benötigt
- Kombination aus Adagrad und RMSprop
 1. Erst moving average des gradienten (mt) und squared gradienten (vt) updaten (das sind die Schätzungen des ersten und zweiten Moments)
 2. Parameter kontrollieren die Abklingraten
default values are 0.9 for β_1 , and 0.999 for β_2

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

3. Moving average beginnt bei 0 \rightarrow bias der moment estimates um 0 herum
Entgegenwirken mit biascorrected estimates

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

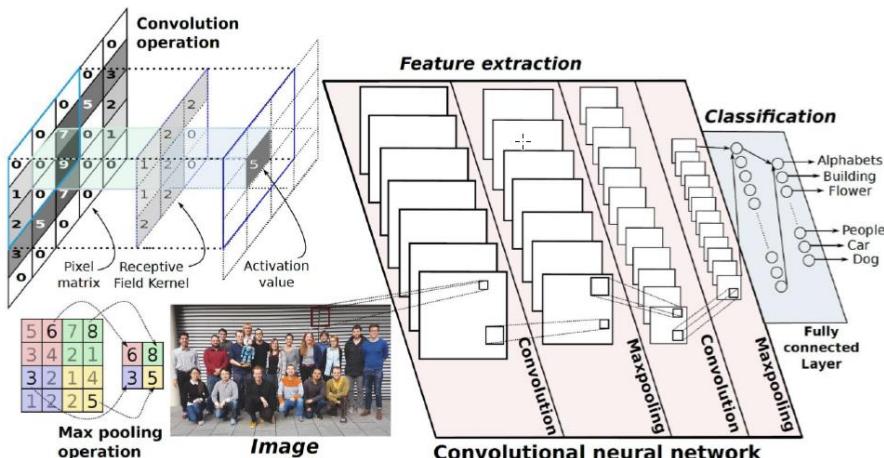
4. Zum Schluss wird upgedatet. Adam ist beliebt, schnell, funktioniert gut, keine Errors der vorherigen Methoden

$$\theta_{t+1} = \theta_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

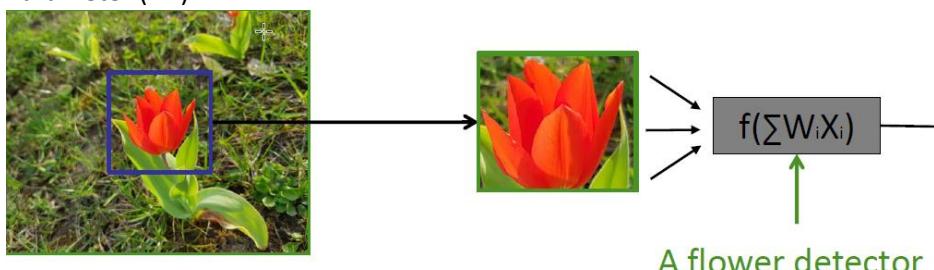
5. CCNs & RNNs

CNN

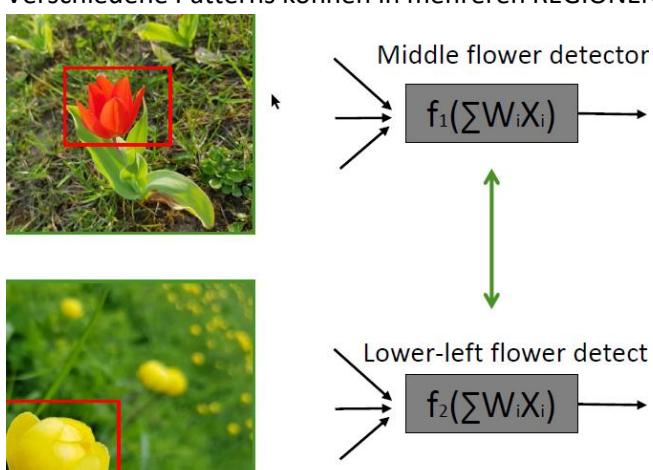
- Generalisierung verbessern, indem man dieselben Gewichte wiederverwendet, um dasselbe Merkmal an mehreren Stellen zu erkennen
- Netzwerk hat convolution Operationen anstatt Matrix Multiplikationen in mindestens einem Layer
- Feedforward
- Benutze dieselben Neuronen für sich wiederholende Faltabläufe (Falten =~ Filtern)
- Reduziert overfitting und führt zu genauerem Model



- In feedforward gibt es tausende Verbindungen und Gewichte, aber eigentlich führen oft nur Schlüsseleigenschaften des Inputs zum entsprechenden Output, also braucht das Netz eigentlich nicht das komplette Signal sondern nur einen kleinen Teil(X_i) davon und weniger Parameter (W_i)

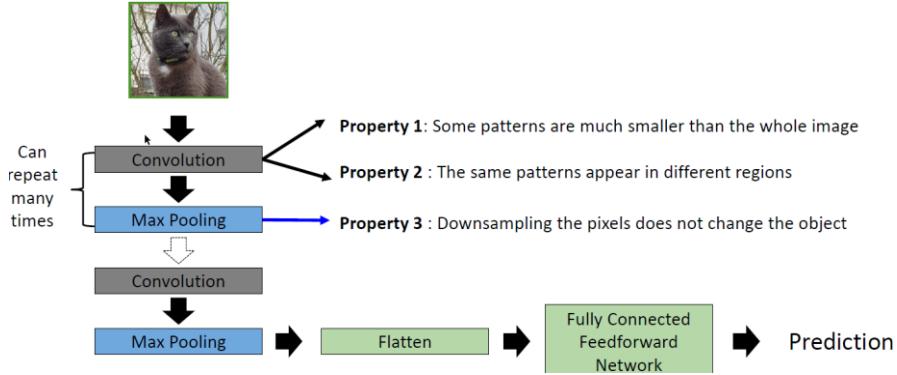


- Verschiedene Patterns können in mehreren REGIONEN auftreten:



- Beide Netzwerke erkennen Blumen, also fast dasselbe. Deshalb können sie dasselbe Parameterset nutzen

- Bilder werden gedownsampling, weniger Parameter zu verarbeiten



Convolution

- Zwei Signale manipulieren um drittes zu formen (* Operator)

$$y[n] = x[n] * h[n] = \sum_{i=-\infty}^{\infty} x[i]h[n-i]$$

$$y[m, n] = x[m, n] * h[m, n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} x[i, j]h[m - i, n - j]$$

- Example im Skript

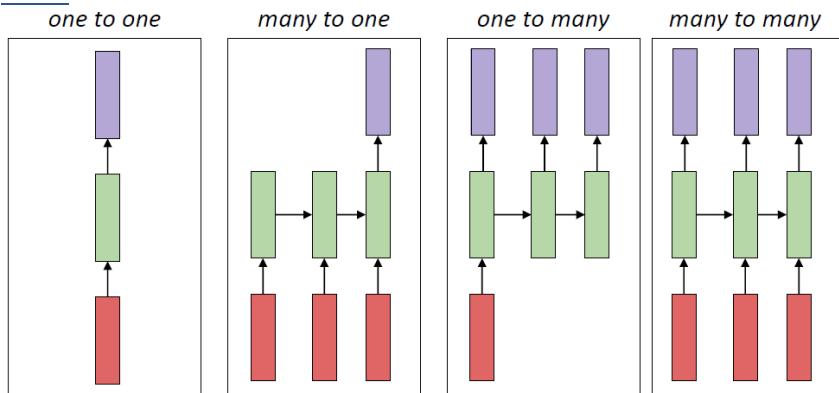
Convolutional Layers

Backpropagation

Max Pooling

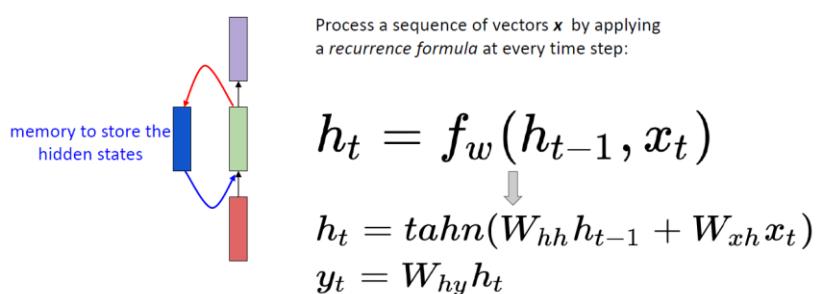
Backpropagation

RNN



- Sequentielle In- und Outputs,
- M21 für Spracherkennung, Wortklassifikation, Videoklassifikation
- 12M für Musikkomposition, Bildlabeling, Textgenerierung
- M2M für Emotionsvorhersage, Spracheverbesserung, Videoklassifizierung auf Framelevel

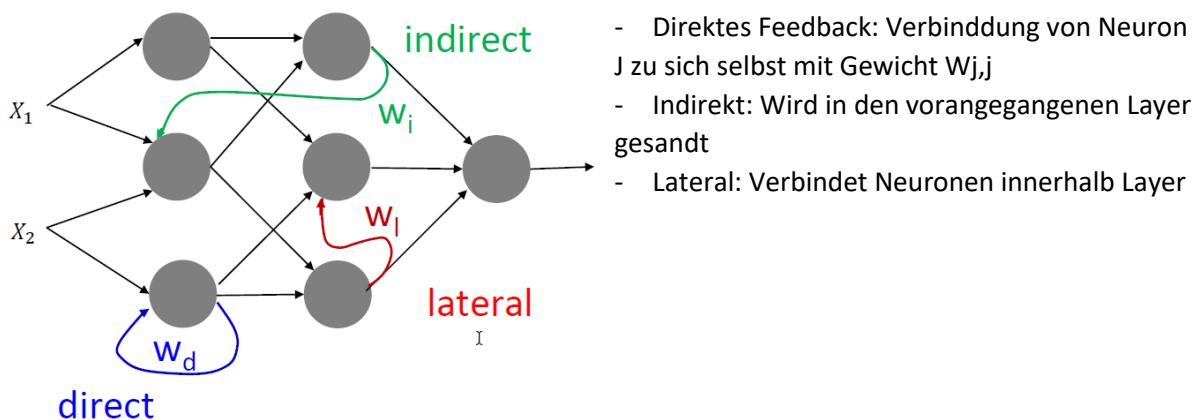
- Output von Hiddenlayer wird in Memory gestored
- Values in Memory als zusätzlicher Input in nächstem Step



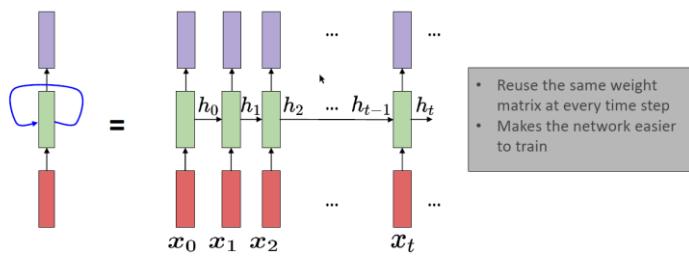
- Fw ist funktion mit Parametern W, xT ist inputvektor, H ist state

Network Structures

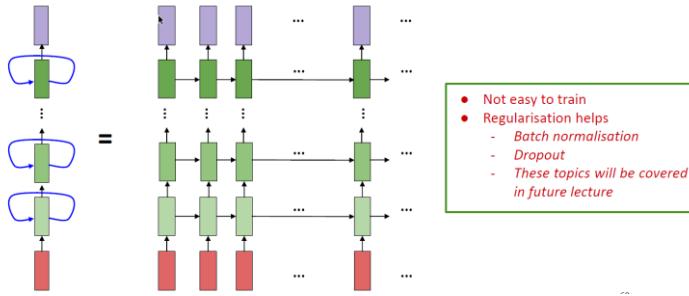
Verschiedene Verbindungen:



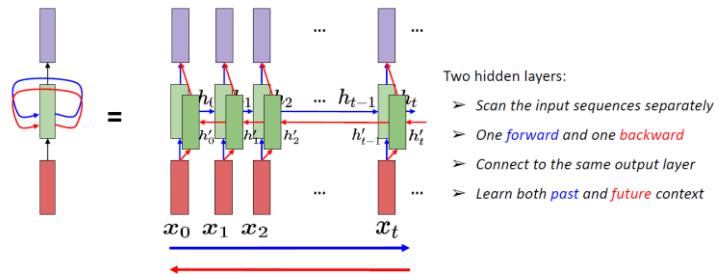
Unrolled RNN



Deep RNN

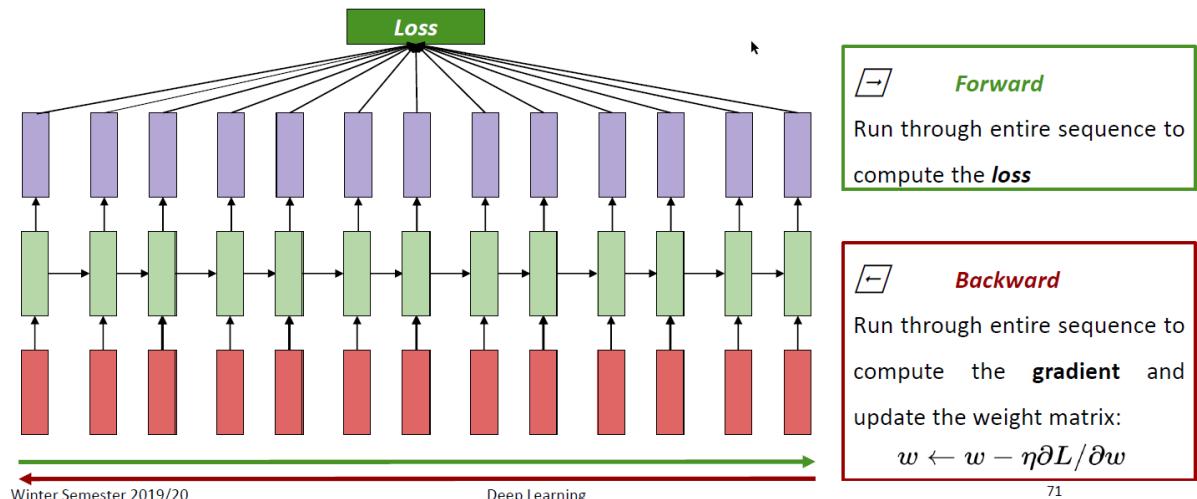


Bidirectional RNN

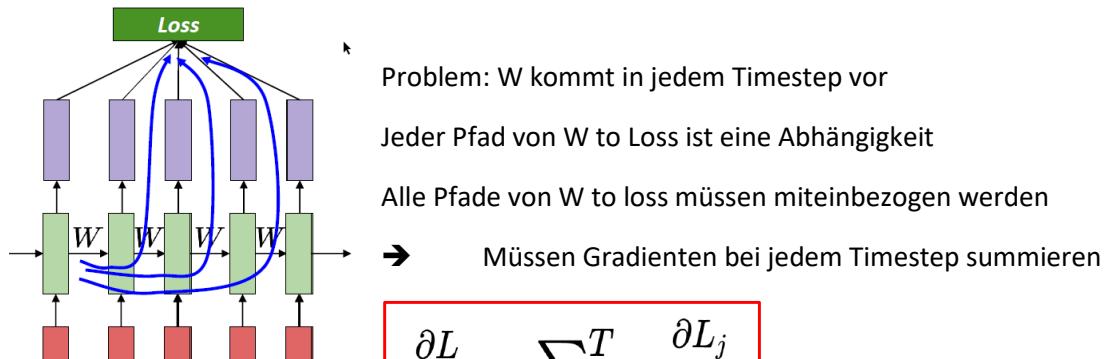


Backpropagation

Backpropagation through time (BPTT)



Gradient flow in simple RNNs



- Dafür nutzen wir mehrere Iterationen der Kettenregel

$$\frac{\partial L}{\partial w} = \sum_{j=0}^T \sum_{k=1}^j \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{t=k+1}^j \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_k}{\partial w}$$

- Wiederholte Matrixmultiplikation führt zu vanishing und exploding Gradienten

Identity-RNN

- Gewichte werden auf Idenditätsmatrix initialisiert
- Aktivationfunctions auf ReLU gesetzt
- Ermutigt Berechnungen nah an Identitätsfunktion zu bleiben
- Gradient baut nicht ab weil Ableitung entweder 0 oder 1 bleibt
- Error wird zurück propagiert

$$\frac{\partial L}{\partial w} = \sum_{j=0}^T \sum_{k=1}^j \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{t=k+1}^j \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_k}{\partial w}$$

$$h_t = h_{t-1} + f(x_t) \rightarrow \left(\frac{\partial h_t}{\partial h_{t-1}} \right) = 1$$

Gated RNN

- Sehr effektive Sequenzmodels
- Pfade erschaffen, deren Ableitungen weder vanishén noch exploden
- Verbindungsgewichte, die sich bei jedem Zeitschritt ändern können
- Alter Status kann vergessen werden
- Netzwerk lernt, wann Status zu säubern ist

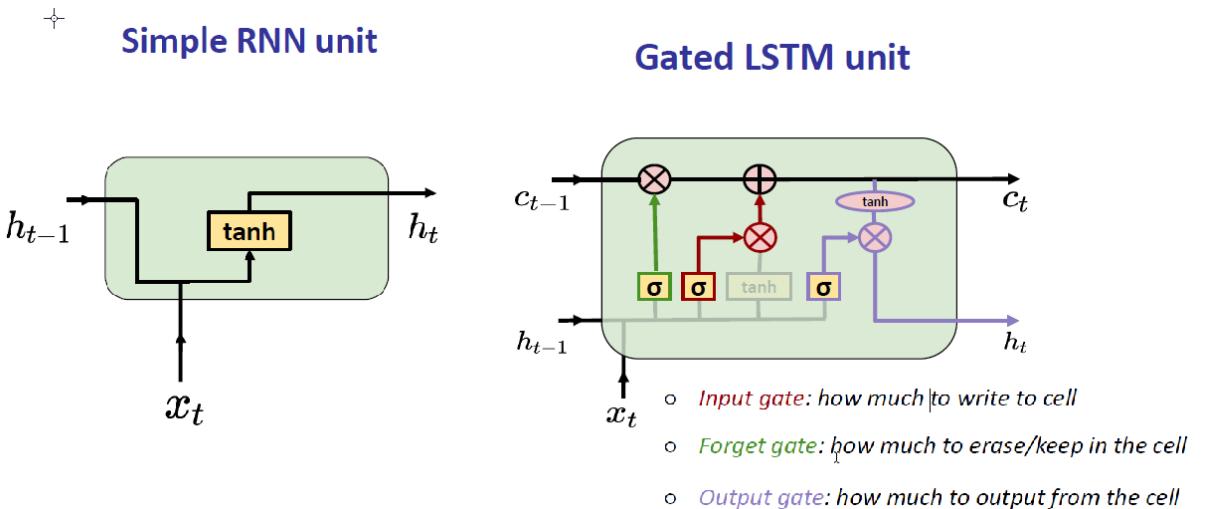
Long Short-Term Memory (LSTM)

- Information über Zeit verwerfen oder behalten
- Tore lernen zu bestimmen was während Training relevant ist
- Informationen werden zu Zelle hinzugefügt oder Removed

1. Forget Gate: Was ist wichtig vom letzten Step zu behalten?

Informationen vom letzten Hidden State und Current Input werden durch Sigmoid-Funktion geschleust. Gibt Wert zwischen 0 (forget) und 1 (keep)

2. Input Gate: Welche Information vom kommenden Step sollte man hinzufügen?
 Updatet den Cell State
 Zuerst wird letzter Hidden State und aktueller Input durch Sigmoid funktion geschleust:
 Nah an 0 heißt Value ist nicht wichtig upzudaten, 1 heißt wichtig
 Dann wird Hidden State und Current Input durch tanh Funktion geschleust, dies hilft das Netzwerk zu regulieren (ergebnis zwischen -1 und 1)
 Dann wird than mit sigmoid output multipliziert: Sigmoid Output entscheidet welche Information vom Thanoutput behalten wird
3. Cell State: Der Networkmemory trägt relevante Informationen während die Sequenz processed wird. Information wird zu Cell State durch Tore hinzugefügt oder entfernt.
 Zuerst wird der Cell State punktweise mit dem Forget Vector multipliziert. Hierbei werden möglicherweise Values fallengelassen wenn es mit Values nahe 0 multipliziert wird. Dann nimmt man den Output vom Inputgate und macht eine Punktweise addition, die den Cellstate zu neuen Werten updatet, die das Netzwerk wichtig findet. Hieraus ergibt sich neuer State der zu neuem Timestep übertragen wird.
4. Output Gate: Was sollte der nächste Hidden State sein?
 Letzter Hidden State und aktueller Input durch Sigmoidfunktion, dann wird der neue Cellstate durch tanhfunktion geleitet. Tanhoutput wird mit Sigmoid multipliziert um zu entscheiden, welche Information der Hidden State carriert sollte. Output ist der hidden State, welcher zum nächsten Timestep übertragen wird



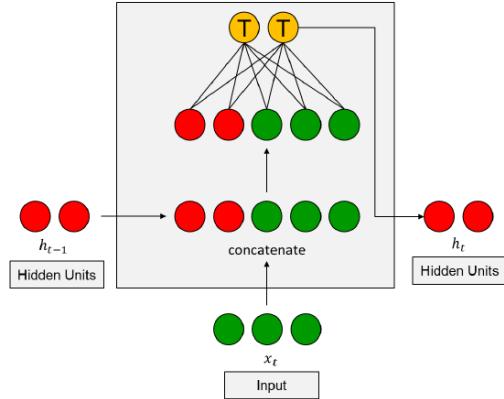
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

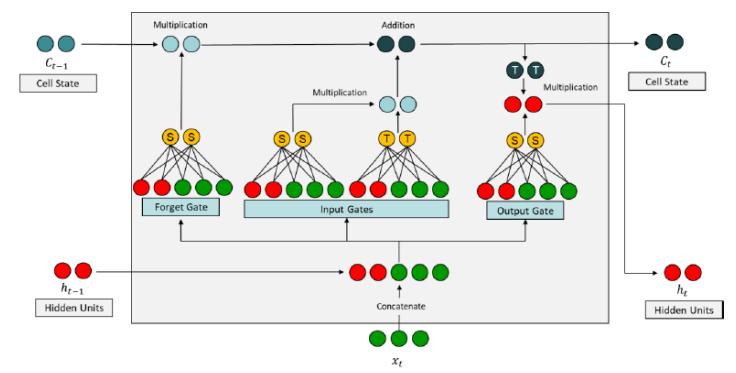
$$h_t = o \odot \tanh(c_t)$$

- Normale RNNs leiden an ShortTermMemory
- Wenn eine Sequenz lang ist können sie keine Information von früheren zu späteren Zeitsteps übertragen, außerdem Vanishing und Exploding Gradients

Simple RNN unit



Gated LSTM unit



- Kann mit Vanishing umgehen, da letzter Cellstate und Input addiert werden. Der Einfluss verschwindet nie, außer das Forgetgate wird komplett geschlossen.

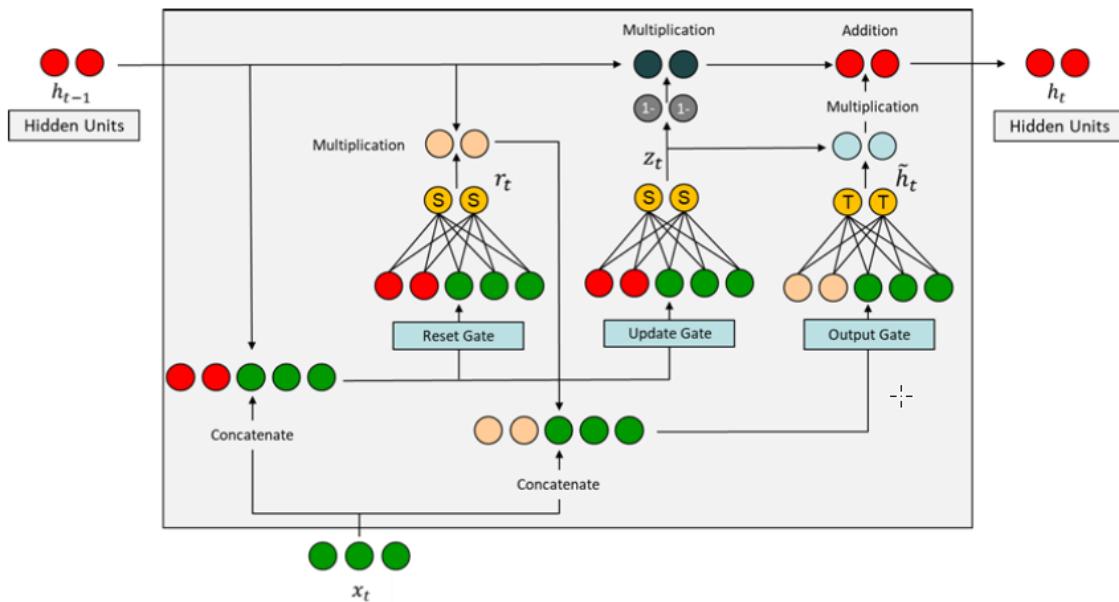
Gated Recurrent Unit

- Vereinfachte LSTM Einheit
- Kein persistenter Zellenstatus
 1. Reset Gate: Wieviel letzte Info vergessen? Wie neuen Input mit prev. Value kombinieren
Info von letztem Hidden State und Current Input wird durch Sigmoidfunktion
0 vergessen, 1 behalten
 2. Update Gate: Wieviel von letztem Memory soll behalten werden?
 - o Kein vanishing issue mehr

Ähnlich wie Forget und Input von LSTM: Welche Info vergessen, welche hinzufügen?

Letzter State und aktueller Input werden zusammen durch S-Funktion: zw 0 und 1
 3. Output Gate
Element-wise multiplication zwischen r_t und h_{t-1} und durch t-FUNKTION
usw siehe Bild

Gated Recurrent Unit



Vergleich zwischen GRU und LSTM

- Beide besser als Simple RNN
- GRU schneller mit weniger Parametern zu trainieren
- Besser als LSTM bei bestimmten Aufgaben
- Gates sind nichts anderes als NNs, die den Informationsfluss regeln

Backpropagation & Regularisation

Backpropagation

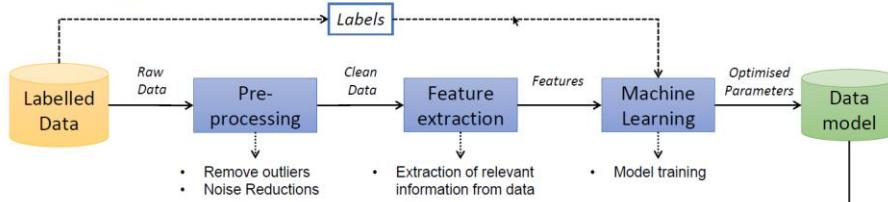
Generalisation

- Eine große Challenge das NN zu überzeugen zu generalisieren und sich nicht nur zu erinnern

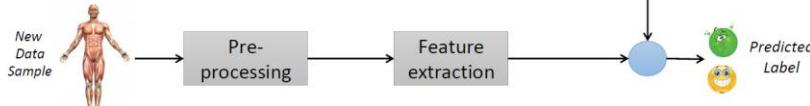
Generalisation of a supervised model

- Man will Training- und Generalisierungserror minimalisieren
- Wenn man mit echten Daten arbeitet ist es unmöglich alles zu observieren, viele Daten sind aufwendig und teuer
 - ➔ Nur begrenzte Anzahl von Training samples, statistische Fehler können auftreten
 - ➔ Generalisierbarkeit muss sichergestellt sein (Fähigkeit, neue Data zu labeln)

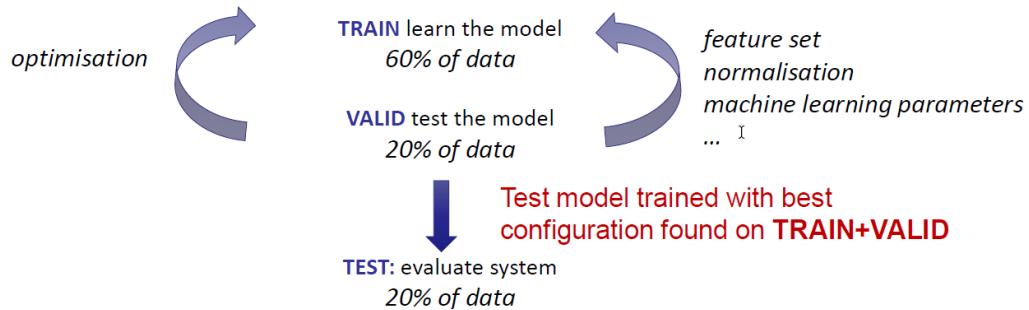
• Training phase – minimise training errors



• Test phase – minimise generalisation errors

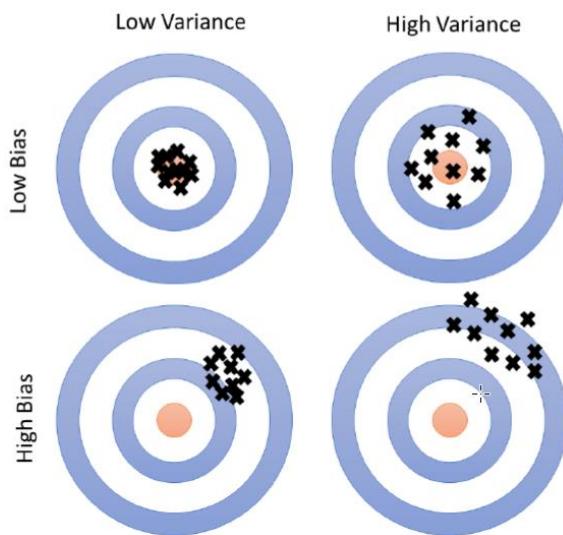


Data Partitioning



- Small dataset: Cross-validation Training
- Randomisiere Speaker ID oder Instanzen
- Teile Dataset in k gleiche Faltungen auf
- Trainiere auf allen (k-1) Falten und Validiere auf Falte k
- Wiederhole k mal um alles zu covern

Bias und Varianzfehler

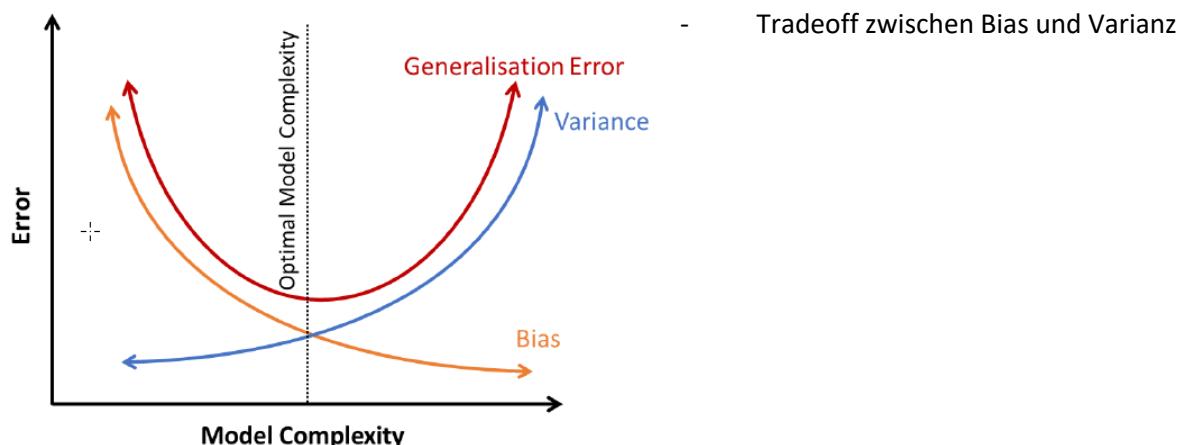


Bias: Wie weit unterscheiden sich die vorhergesagten Werte im Schnitt von den echten Werten?

Varianz: Wie unterschiedlich wird die Vorhersage des Models sein wenn man unterschiedliche Stichproben derselben Population nimmt

Generalisation Errors

Minimalisieren



Underfitting

- High bias und keine Sensitivität bzgl. Datavariation
- Lernt das Problem nicht ausreichend
- Leicht zu beheben: Modelkomplexität erhöhen, mehr Funktionen um input auf output zu mappen
- Mehr Layers/nodes

Overfitting

- Versucht ALLES an Variation zu finden
 - Zu viel gelernt: Gut im Trainingdataset, schlecht bei neuen Daten
 - zB Hund mit Kissen vs einfach nur Hund
 - Problem ist die Noise, wir brauchen nur das Signal
- Regularisation

Regularisation

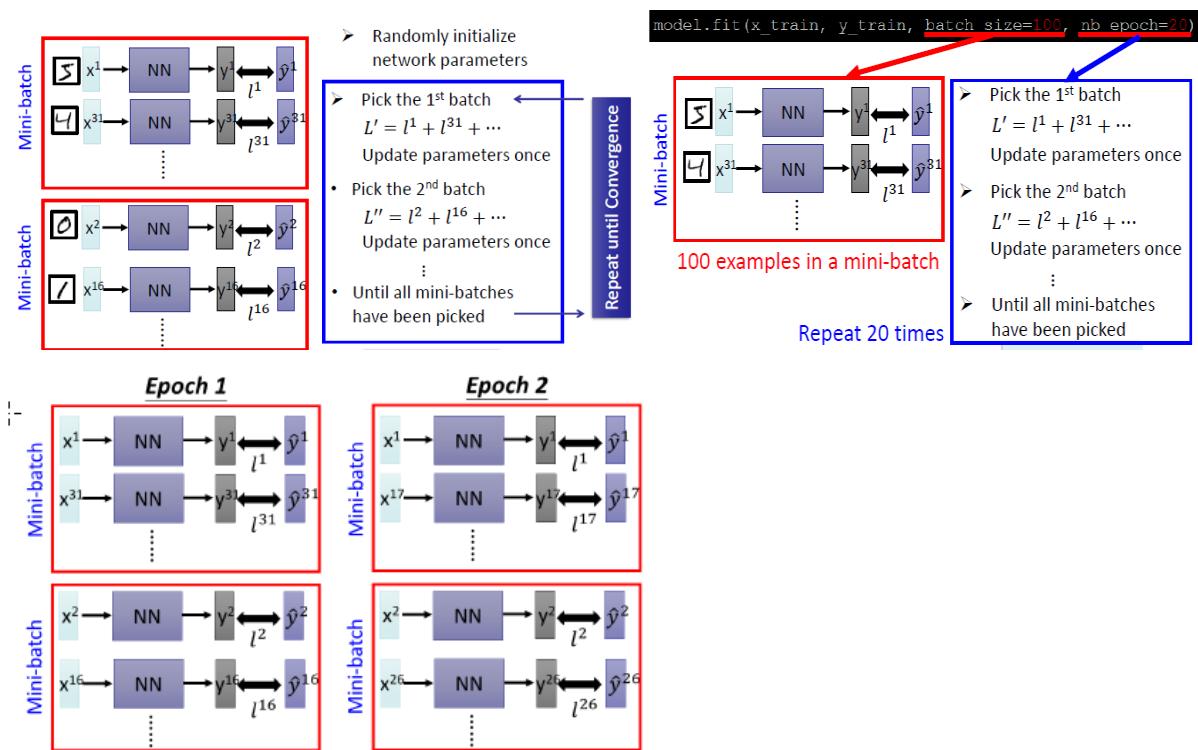
- Dazu designed, Generalisierungerror zu verringern, möglicherweise auf Kosten des Trainingerrors

Mini-Batch Learning

- Update für jede Batch von n Trainingsbeispielen
- Reduziert Noise in Varianz der Gewichtupdates

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J(\theta^{(t)}; x^{(i:i+n)}; y^{(i:i+n)})$$

- Vorteile: Stabile Konvergenz, Gute Lernrate, Gute Vorhersage des Minima
- Nachteil: Gesamtloss nicht summiert
- Batchsize muss groß genug sein um gute Einschätzung des Gradienten zu geben, klein genug um angemessene Noise in Update zu geben
- Eine Epoche ist ein Trainingsdurchgang über Datensatz, besteht aus N (forward pass + backpropagation) mit n = Batchanzahl



Weight Penalties

- Kostfunktion bekommt Weight Penalty Term
- $\tilde{J}(\theta; x, y) = J(\theta; x, y) + \alpha \Omega(\theta)$
- Große Gewichte machen das Netzwerk instabil, weil dann kleine Variation oder noise im Input große Outputunterschiede hervorbringen
- Model soll so den Input auf Output mappen, dass die Weights gering bleiben

$$\alpha\Omega(\theta) = \|\theta\|_1$$

- L1 Norm: Summe absoluter Weightwerte, mutigt Gewicht dazu an 0 zu sein, resultiert in niedrigen und null Gewichten

$$= \sum_n |\theta_n|$$

$$\alpha\Omega(\theta) = \frac{1}{2} \|\theta\|_2^2$$

$$= \sqrt{\sum_n |\theta_n|^2}$$

- L2 Norm: Summe der quadrierten Weights, bestraft große Gewichte

Einfache lineare Regression als Lerntechnik:

- Actual (observed) values $y = \theta^T X + b$
- Predicted Value: $y_p = \theta^T X + b$
- Wish to learn θ (and b)
- Set Cost function to square of ℓ^2 norm of error

$$C(\theta) = \|e\|_2^2$$

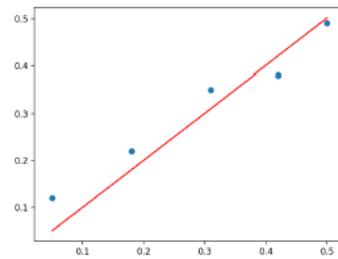
$$= \|X\theta - Y\|_2^2$$

$$= (X\theta - Y)^T (X\theta - Y)$$

Calculate gradient, set to 0 and solve

$$\nabla C(\theta) = 0$$

$$\hat{\theta} = (X^T X)^{-1} X^T Y$$



Einfügen der Gewichtstrafen:

- Standard Cost function:

$$C(\theta) = \|e\|_2^2 = \|X\theta - Y\|_2^2$$
- Insert regularisation term to penalizes the overall cost function in case the magnitude of the model parameter vector is high

$$L(\theta) = \|X\theta - Y\|_2^2 + \lambda \|\theta\|_2^2$$
- Taking gradient of $L(\theta)$ and setting to zero gives:

$$\hat{\theta} = (X^T X - \lambda I)^{-1} X^T Y$$
- A higher value for λ would therefore result in smaller values of $\|\theta\|_2^2$ thus making the model simpler and prone to high bias

Data Augmentation

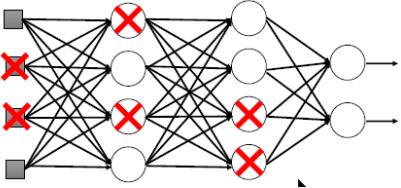
- Mit Fakedata trainieren(mehr Data erschaffen), damit Model besser trainieren kann: Bilder flippen, zoomen usw

Training with Noise

- Noise hinzufügen, damit sich das Netzwerk keine samples „merkt“
- Gibt niedrigere Gewichte und niedrigeren Generalisierungssrror

Dropout

- Mehrere verschiedene Netzwerke werden alle verschiedene Noise overfitten
- Deshalb mehrere leicht verschiedene Netzwerke zusammenwerfen und mitteln
- Hierzu werden während des Trainings zufällig nichtoutput-Units entfernt



- Jedes Neuron hat p% rauszufallen, vor Training ausgesucht, Netzwerk ist geändert, Training wird fortgesetzt, für jedes Minibatch resamplet man die Dropout Neuronen
- Während Testtime werden outgoing Weights mit P multipliziert

Early Stopping

- Wenn man zu lange lernt hört Model auf das Signal zu lernen und fängt an Noise zu lernen
- Lang genug lernen für Mapping, kurz genug um nicht overzufitten
- Trainingsstop wenn Generalisierungssrror wächst
- Nach jeder Epoche wird Model mit holdout validation Datensatz geprüft
- Wenn die Performance hierbei sinkt, wird Training gestoppt
-

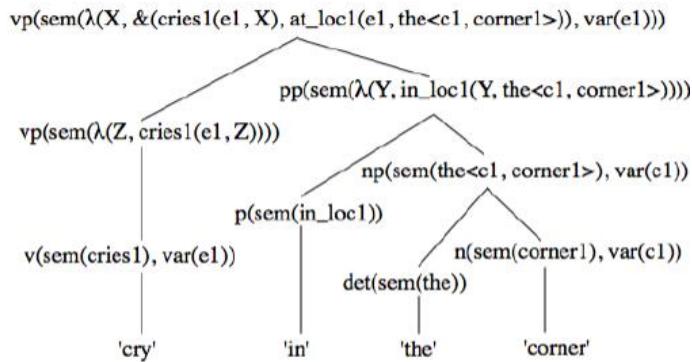
Sequence To Sequence Learning

Natural Language Processing

- Absichtliche und unabsichtliche (Mimik interpretieren zB) Kommunikation
- Jeder Satz benötigt Interaktion zwischen folgenden drei Sprachkomponenten:
 1. Inhalt: Sprachstruktur, Bedeutung, Vokabeln, Semantik
 2. Form: Sprachbedeutung, Grammatik, Syntax
 3. Nutzen: Ziele und Soziale Aspekte, verschiedene Wortkombinationen aussuchen
- NLP hat es schwer, denn Sprache ist vielfältig, mehrdeutig, kontextuell, situationsbedingt
- Ein Symbol kann mehrere Sachen bedeuten, Vokabular ist riesig

Discrete Language Models

- Konventioneller Ansatz menschliche Sprache zu erlernen, zB:
- Wortbedeutung, Morphology lernen (Wie sie geformt sind und zu anderen stehen)
- Kontext der Erscheinung
- Semantisches Verständnis, grammatischen Komposition, Satzbaum
- Fakten: Regex, Regeln



- Zuerst versuchte man Worte zu Vektorisieren, jedes Wort hat eigene Kennung, aber Vektoren werden riesig und haben keine Verbindung zueinander
- Aber: Benötigt viel menschliche Arbeit
- Sachen werden vergessen, nicht geupdated
- Subjektiv und verschwommen
- ➔ Deshalb kontinuierliche Darstellung von Wörtern verwenden

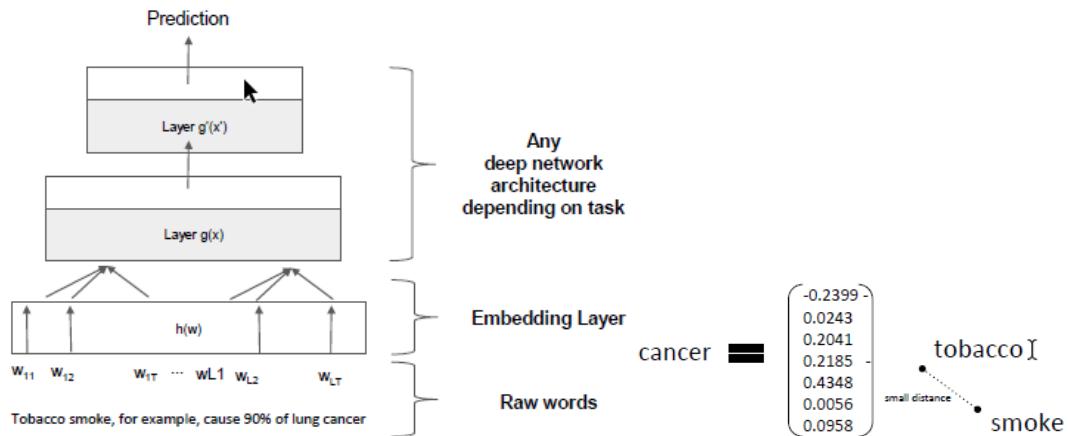
Continuous Space Language Models

1. Distributional Hypothesis:
Sprachelemente mit ähnlichen Verteilungen haben ähnliche Bedeutungen.
2. Vector Space Model:
In einen kontinuierlichen Vektorraum eingebettete Wörter sind semantisch nebeneinander eingebettet.

Entscheidend zum Verstehen eines Wortes ist, in welchem Kontext es erscheint. 2 Wege diese Prinzipien umzusetzen:

- Zählbasierte Methoden: In großem Text wird gezählt, wie oft ein Wort neben Nachbaren erscheint und auf Vektor runtergebrochen (zB Latent Semantic Analysis)
- Predictive methods: Ein Wort direkt anhand seiner Nachbaren vorhersehen, anhand kleiner, dichter, eingebetteter Vektoren (zB Neural Probabilistic Language Models)

NLP with Deep Learning



Word Embedding

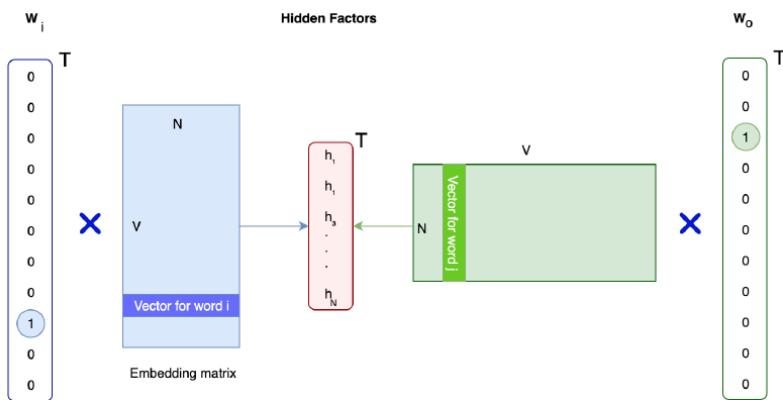
Wir wollen:

- einen dichten Vektor (Meiste Values nicht 0)
- andere Worte im Kontext vorhersehen
- andere Worte sollen auch von Vektor repräsentiert sein
- Gleichheitsmessungen zwischen Vektoren benutzen

Word Embeddings sind Erfolgsrezept bei jedem NLP Model

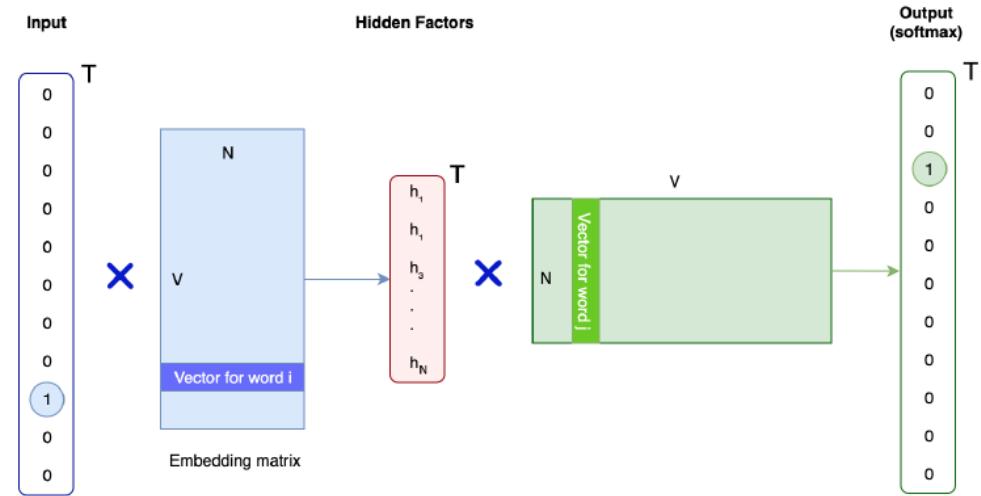
Dense Representation

- Transformiere one-hot Vektor in denses Repräsentation h mit weniger Dimensionen
- Gleiche Worte sollten nah zueinander sein im projizierten Raum



The Simple Approach

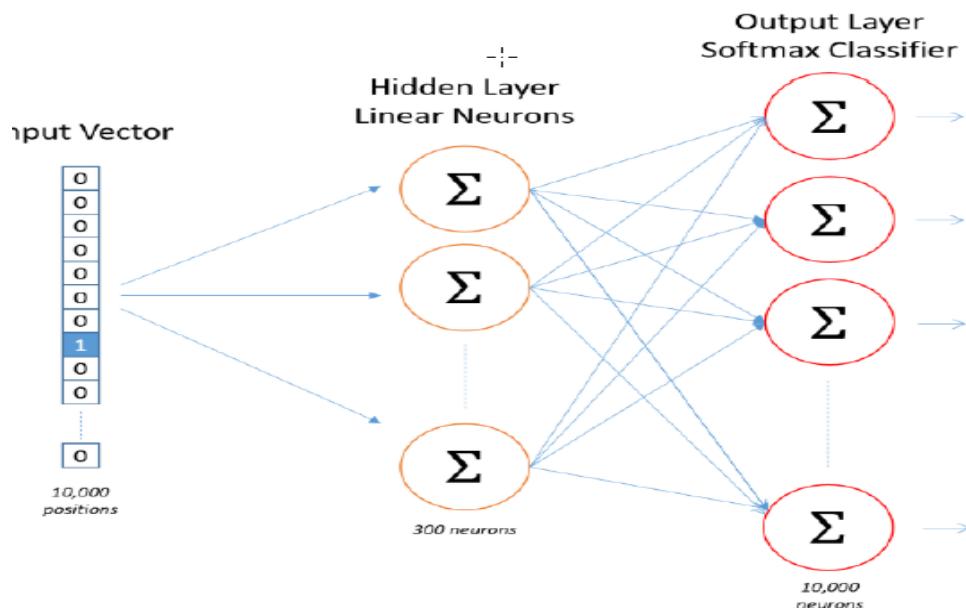
- Fange mit Wort W_j an
- Zuerst die Vektorrepräsentation mit eingebetteter Matrix ausrechnen
- Mit anderer Matrix multiplizieren um ein ähnliches Wort w_0 vorherzusagen
- Softmaxfunction verwenden um Outputwahrscheinlichkeiten zu kalkulieren



Netzwerk Architektur

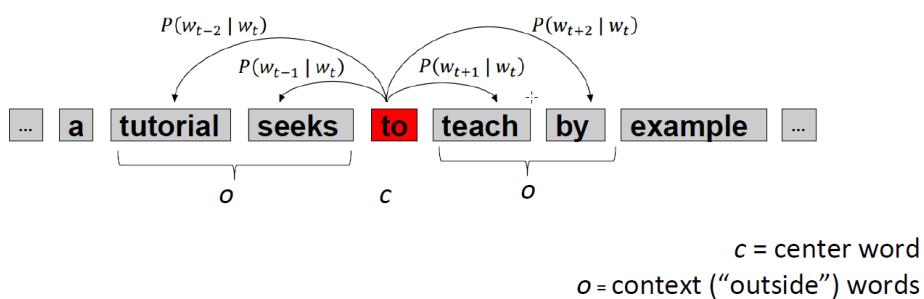
Training Data

- Input ist Vektor von Wort
- Output ist Vektor von Outputwort
- Trainierbare Parameter ist die Gewichtsmatrix der Hidden Layers: Eine Reihe = ein Wortvektor



Word2Vec

Effizientes Vorhersagemodel um Wortembeddings aus Text zu lernen



- Skipgrammodell produziert eine Vorhersage pro möglichem Nachbar
- Objective function: loglikelihood der vorhergesagten Worte mit gegebenem Zielwort t

$$\mathcal{L} = \frac{1}{T} \sum_{t=1}^T \sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} | w_t)$$

- Training Words
- Wie berechnet man $p(w_0/w_t)$?
 1. Suche die entsprechenden Reihen und Spalten, die auf w_i und w_0 zeigen, in der entsprechenden Matrix
 2. Benutze Softmaxfunktion um die conditional probability zu berechnen

Word similarity measure^Δ
using dot product

$$p(w_o | w_t) = \frac{\exp(v'_{w_o} v_{w_t})}{\sum_{w=1}^W \exp(v'_{w'} v_{w_t})}$$

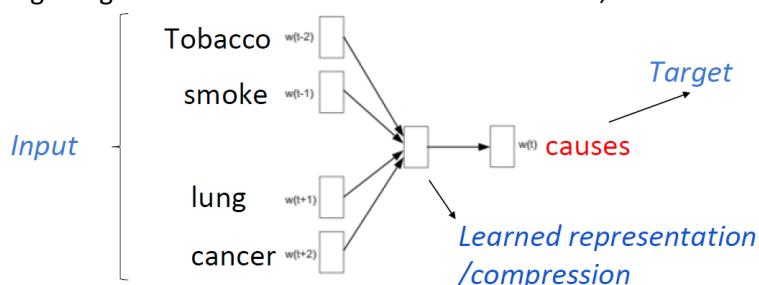
↑ Sum all possible similarities together
to normalise into probability distribution

- Algorithmus:
 1. Definiere Wortfenster m, das linke und rechte Wörter als context o beinhaltet
 2. Durch jede Position t im Text gehen, die Centerword C und context o hat
 3. Benutze die Gleichheit der Wortvektoren für C und o um die Wahrscheinlichkeit $P(w_{t+j} | w_t)$ zu berechnen
 4. Passe die Wortvektoren weiterhin an um diese Wahrscheinlichkeit zu maximieren
- Wege, die Performance von Wordembeddings zu messen:
 1. Word Vektor Analogies:
 2. Intrinsic Word Vektor Analogies:
 3. Extrinsic Evaluation:
 4. Computation Costs

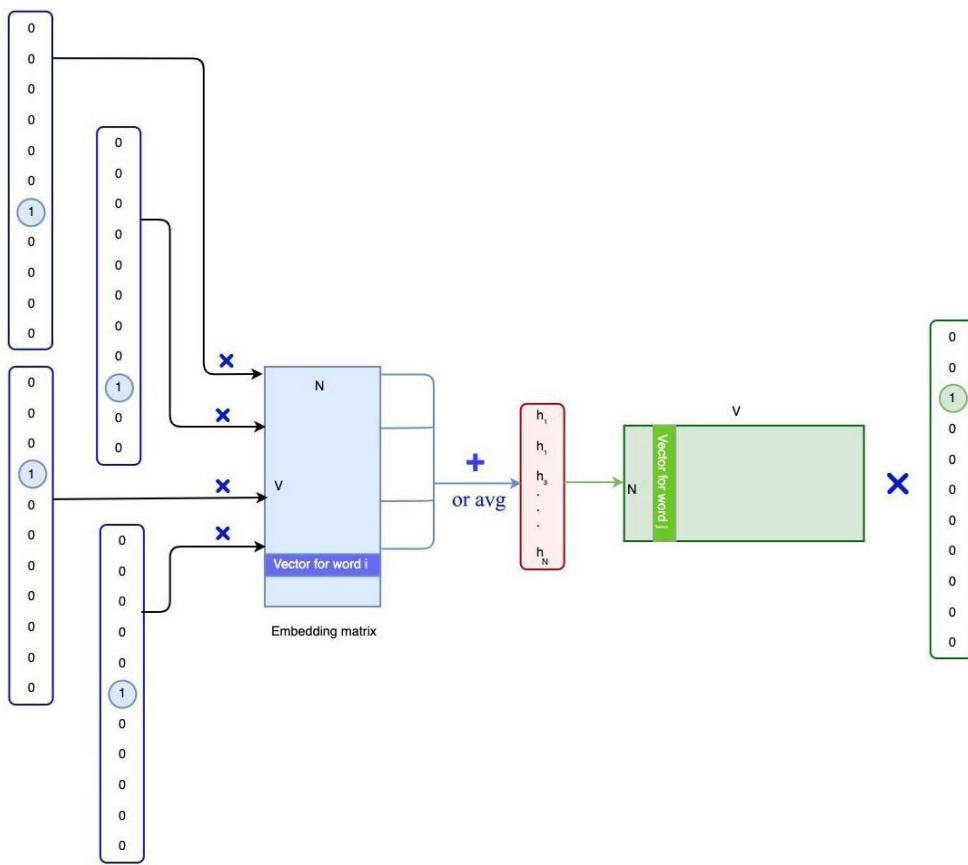
Andere Word2Vec Models (Außer Skipgram)

1. Continuous Bag-Of-Words model (CBOW) ist Spiegel von Skipgram:

- Sagt Targetwords von Sourcecontextwords vorher, man kennt Context und will Zielwort



- Schneller als skipgram, weniger comp. Komplexität
- Modelliert keine seltenen Worte



2. Global Vectors for Word Representation (Glove)

- Hybrid aus count-based und predictive
- Training auf globalen Wort-Wort co-erscheinungs statistiken angesetzt
- Schnelles Training, weil die Anzahl der nicht-null Matrizeneinträge klein ist, weil die globalen Statistiken vorberechnet sind

Praktische Anwendung

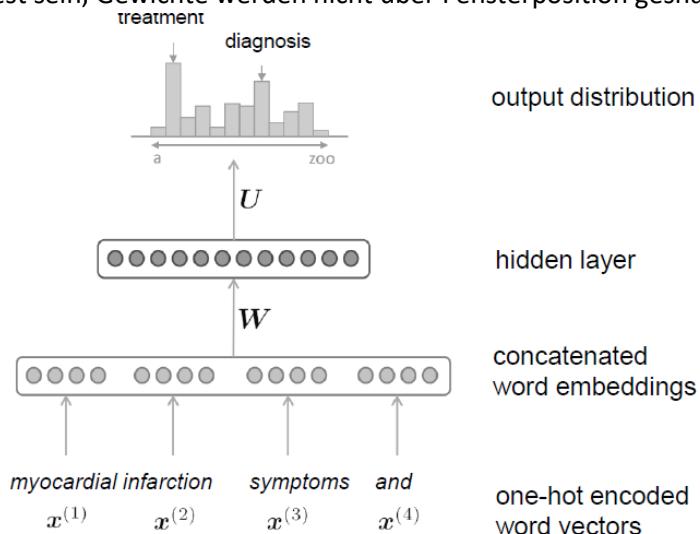
Wie WordEmbedding nutzen, um NLP Problem zu lösen?

1. Off-the-shelf models nutzen
 - Word2Vec (Google) Glove(Standford) fastText(Facebook) sind 3 SOTA word embeddings
 - Gewichte von embedding layer einfrieren und neues, trainierbares Layer auf spezifischem Problem einfügen
 - Ein Wortvektor für bestimmtes Problem umzuschulen ist hart, weil sich nur die Worte im Trainingsset ändern, nicht aber die similar words -> Syntax und Semantik zerstört
2. Domänenspezifisches Wordembedding bauen
 - Spellingchecker nutzen
 - Trainingsdata erweitern, indem man frei erhältliche Data wie Wikipedia 2014 oder Gigaword 5 oder Twitter nutzt-
 - Unterliegende WordEmbeddings sind oft wichtiger für gute Performance als das Model zu tunen

Neural Language Models

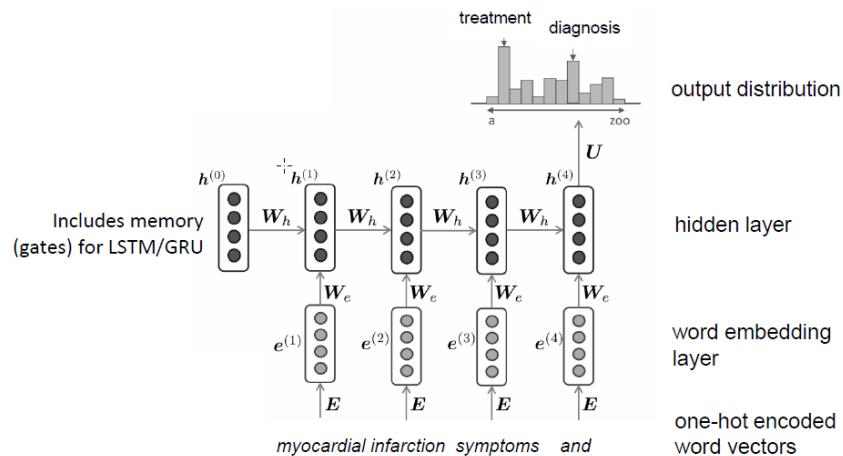
Feedforward

- Feed Forward NN um nächstes Wort in Satz vorherzusagen
- N-gram ist bewegendes Fenster aus Worten mit Festgröße, Wörter können nur klein und fest sein, Gewichte werden nicht über Fensterposition geshared

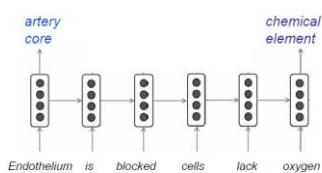


RNN

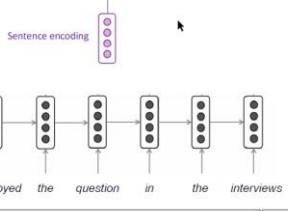
- Auf gesamten Corpus trainiert und von Stochastischem GD optimiert
- Achtet auf Wortfolge, jede Inputlänge kann verwendet werden, Gewichte werden über Zeitsteps geshared
- Gut für Sequenzklassifikation, weniger Text, nicht parallelisierbar deshalb slow



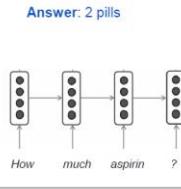
Named Entity Recognition



Sentiment classification

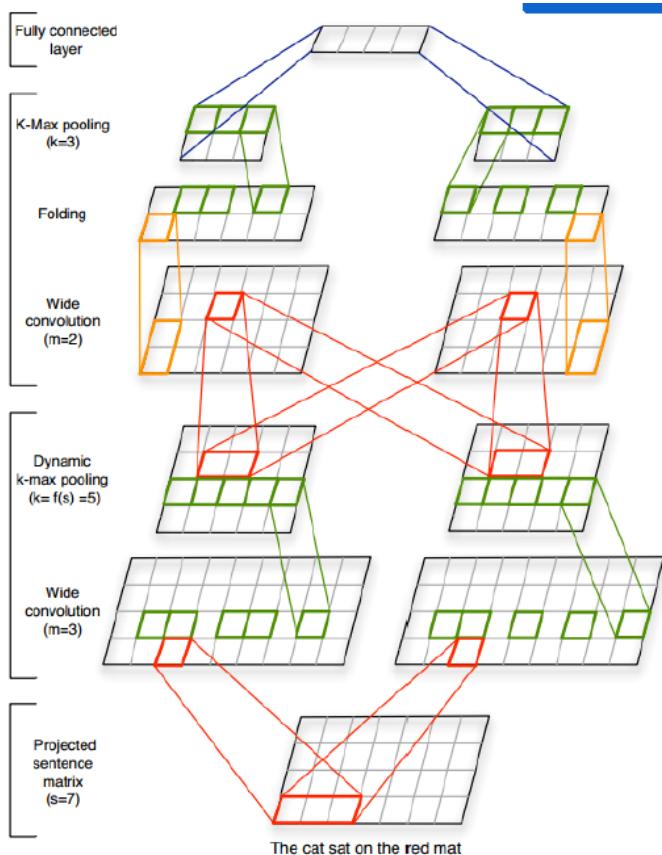


Question answering



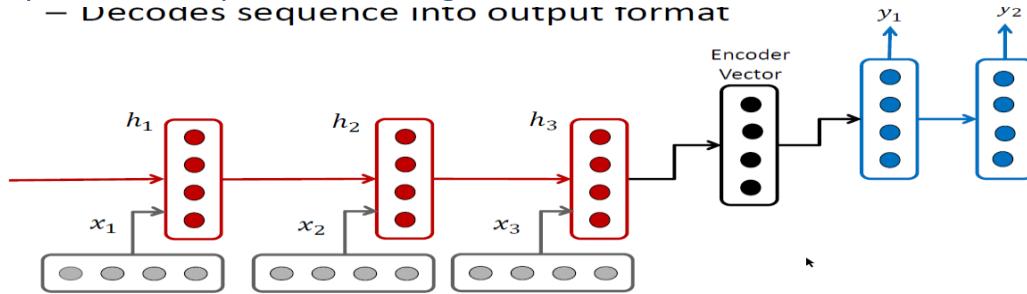
CNN

- Purer DL Ansatz, der linguistisches Wissen ignoriert
 - Gut für Text classification
 - Auf GPU parallelisierbar
 - Mehrfach Vektoren für jede mögliche Phrase parallel rechnen
1. Initialisiere mit vortrainierten Wortvektoren (word2vec d=300)
 2. Verkette die Vektoren jedes Wortes zu Satzvektor
 3. 1d-convolution: Mehrere Fenstergrößen kreieren 3,4,5grams von Sätzen
 4. Nur die wichtigsten Activations von maxc oder averagepooling erfassen
 - Man muss zwar Dropout verwenden, aber trotzdem sehr effizient und einfach



Sequence to Sequence Learning

– Decodes sequence into output format



- Die Quellsequenz zu Vektor codieren
- Sequenz zu Outputformat decodieren

Encoder

- Eine Reihe von recurrent Units (LSTM oder GRU) (die 3 roten im Schaubild)
- Jede nimmt ein einzelnes Element der Inputsequenz an, sammelt Informationen darüber und leitet es weiter
- Hidden States werden mit $h_t = f(W^{hh}h_{t-1} + W^{hx}x_t)$ berechnet
- Normale RNN, in denen wir Gewichte zum vorherigen Hidden State und neuem Inputvektor anwenden

Encoder Vektor

- Finaler Hiddenstate, berechnet von der obigen Formel
- Zielt darauf ab, Informationen für alle Inputelemente einzufangen, um dem Decoder dabei zu helfen genaue Vorhersagen zu treffen
- Ist der initiale Hidden State vom Decoderpart

Decoder

- Reihe an recurrent Units, jede sagt Output y_t an Zeitstempel t vorher
- Jede akzeptiert einen Hidden State von der vorangegangenen Unit und produziert Output und eigenen Hidden State

$$h_t = f(W^{hh}h_{t-1}) \quad y_t = \text{softmax}(W^{yh}h_t)$$

- Softmax normalisiert Vektor in Wahrscheinlichkeitsverteilung

S2S kann unterschiedliche Längen zusammenmappen, In und Output hängen nicht zusammen

Training

- Decoder gezwungen korrekte Sequenzen zu generieren
- Wird bestraft, wenn eine Sequenz mit niedriger Wahrscheinlichkeit zugewiesen wird
- Loss für jeden Output berechnet und über Outputsequenz aufsummiert
- Neue Parameter mit GD berechnet
- Typische Lossfunction ist cross-entropy
- Bei jedem Output produziert das Netzwerk eine Wahrscheinlichkeit aller möglichen Outputs
- Crossentropy bestraft Unterschiede zwischen Verteilungen

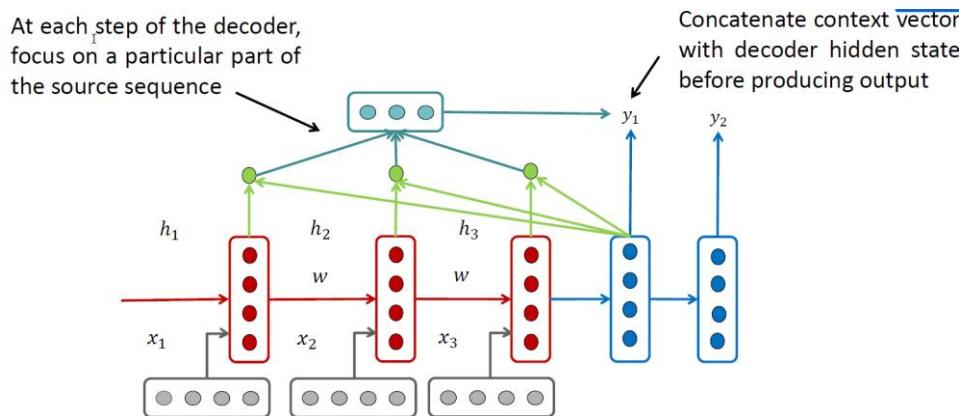
Attention Mechanisms

Bei S2S ist Encoder Vektor Bottleneck:

- Herkömmliche Technik verwirft alle Zwischenencoderstates und nimmt nur finalen um den Decoder zu initialisieren:
- Klappt nur gut für kleine Sequenzen
- Wenn Sequenzen länger werden, wird Vektor zum Bottleneck, weil es schwer wird lange Sequenzen in einzelnen Vektor zu packen

Bei AM werden alle States benutzt, um neuen Kontextvektor zu bauen

- Wahrscheinlichkeitsverteilung mappt jeden Input auf den Outputstate den der Decoder generieren will
- Dieser Vektor wird benutzt, wenn die Outputsequenz codiert wird
- Decodes sammelt also globale Informationen, verlässt sich nicht nur auf einzelnen Hiddenstate
- Während dem Training lernt das Netz, welche Inputs für den Task wichtig sind



Core Steps

- Es gibt encoder HS
- An jedem Timestep gibt es decoder HS
- Man ermittelt AttentionScore für diesen Step
- Man verwenden Softmaxact um die Attention-Verteilung für diesen Step zu bekommen
- Wir nutzen diese um eine gewichtete Summe aller Encoder HS zu nehmen um Attention Output zu bekommen
- Am Ende verkettet man den Attention Output mit Decoder Hidden State

Hierarchical Attention

- Dokument klassifizieren, das aus vielen Sätzen besteht
 1. RNN auf Wortlevel
 2. Attentionmodel benutzen um Worte zu extrahieren, die für Satzbedeutung wichtig sind
 3. Diese Repräsentationen aggregieren, um Satzvektor zu formen
 4. Selben Prozess auf abgeleiteten Satzvektor anwenden, um Vektor zu generieren, der Dokumentbedeutung misst
 5. Dieser Vektor wird für die finale Klassifizierung genutzt

Connectionist Temporal Classification

- Sequence learning auf unsegmentierte Daten anwenden
 - Vorsegmentierung ist zeitaufwändig, teuer, und meist unmöglich
-
- In jedem Schritt kann Netzwerk ein „blank“ Label oder jeden beliebigen Charakter im Vokabular L ausgeben
 - Transformiere die Outputs in conditional Wahrscheinlichkeitsverteilung über labnel Sequenzen, womit man die Wahrscheinlichkeit für jeden Pfad berechnen kann
 - Die gesamte Wahrscheinlichkeit jeder Labelsequenz kann gefunden werden, wenn man die wahrscheinlichkeiten der unterschiedlichen Pfadlängen, die zur Sequnz führen, aufsummiert

Pfad

- Pfad P ist Labelsequenz auf framlevel
 - Likelihood kann in unabhängige Frames geteilt werden
- $$Pr(p|X) = \prod_{t=1}^T y_t$$

Training

Decoding

Deep Reinforcement Learning

Introduction

Wiederholung:

Supervised Learning

- Lernt auf labeled Data
- Bekannte Trainingdata als Input
- Vorhersagen wie Klassifikation oder Regression als Output
- Gute Performance, falls große Menge an labeled
- Labeled Data ist zeitaufwändig und nicht für jeden use-case brauchbar

Unsupervised Learning

- Lernt auf unlabeled Data
- Input ohne Anmerkung
- Output clustert die Daten

Reinforcement Learning

Case: Interactive agent

- Input ist Umgebung
- Output ist Aktion, mögliche Aktionen sind gegeben
- Vorgefertigtes Ziel, zB Straße überqueren

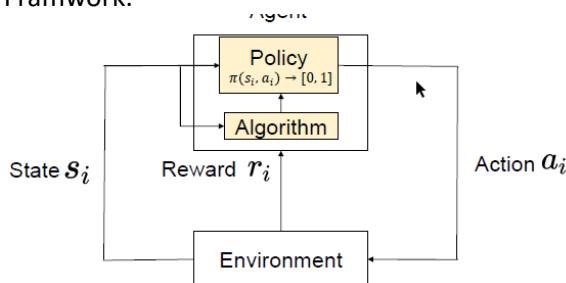
Ansätze

- Unsupervised: Nur Clustering von Umgebungsdaten möglich, Clustering beim überqueren
- Supervised: Die beste Aktion für jeden Datapoint, sehr zeitaufwändig, evtl. viele Aktionen die zu Ziel führen, vom Lehrer lernen, Label move und label wait bei Straße überqueren, lerne statisches Model um Label zu predicten
- Reinforcement: Agent erkundet Umgebung, bekommt Belohnung für jede Aktion, Agent lernt, beobachtet Straße und lernt

Reinforcement Learning

Basics

Framework:



State:

- Umgebungs und Agentenbeschreibung, handgemachte Features, momentaner State s_i von momentaner Situation extrahiert

- zB Koordinaten von SuperMario, Distanz zum nächsten Hinderniss

Action:

- zur Verfügung stehende Aktionen, zB Springen, führt zu neuer Situation s_{i+1}

Policy:

- Wahrscheinlichkeit a_i in Situation s_i auszusuchen
- Ziel während Training lernen
- Wird jeden Zeitstep berechnet

Belohnung:

- R_i
- Kann von Situation s_{i+1} abhängen
- Bei jedem Zeitstep, Basis für Zielerlernung

Problem:

- Statetransitions und Belohnungen schwer vorherzusagen
- Nicht deterministische Umgebung: Gleiche Aktion bei verschiedenen Zeitpunkten resultiert in anderem Reward und State

Exploitation:

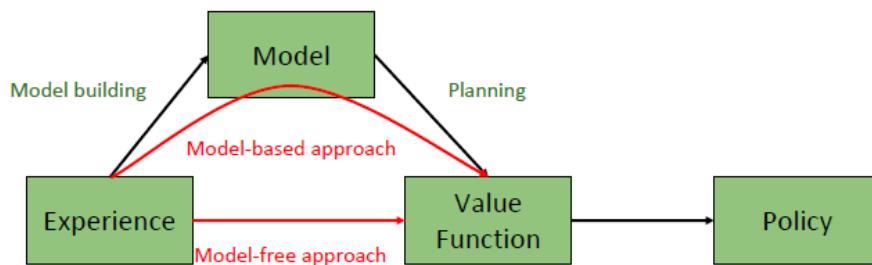
- Vielversprechendste Aktion aussuchen
- Garantiert hoher Reward (falls gut erforscht)
- Keine neue Information

Exploration:

- Weniger erforschte Aktion aussuchen
- Möglicherweise niedriger Reward
- Informationsgewinn

Ziel:

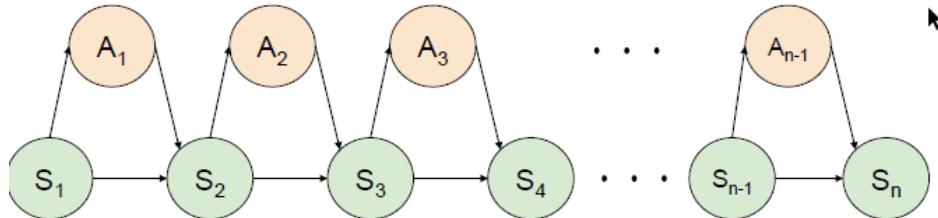
- Erwarteter künftiger Reward v_{pi} maximieren, hängt von Policy ab
- Policy lernen um reward zu maximieren
- Problem ist unendliche Runtime des Algorithmus und somit die unendliche Summe an Rewards



Algorithmen

Markov Decision Process MDP

- Problem modellieren
- Definiert von Stateset S , Actionset A , Rewardfunction $R(s_i, a_j)$
- Transitionfunktion $T(s_i, a_j, s_l) \rightarrow 0,1$
Wahrscheinlichkeit um von State nach State zu gehen wenn A ausgesucht wird



- Optimal value function spezifiziert die bestmögliche Performance in MDP, optimaler Reward von State. MDP ist gelöst wenn wir diese Funktion kennen
- Lerne den optimalen Value per iterieren:


```

initialize V(s) arbitrarily
loop until policy good enough
  loop for s ∈ S
    loop for a ∈ A
      Q(s, a) := R(s, a) + γ ∑s' ∈ S T(s, a, s')V(s')
      V(s) := maxa Q(s, a)
    end loop
  end loop

```
- Leite optimale Policy von optimalen Values ab
- Problem ist, dass das nur funktioniert wenn Model bekannt, deshalb muss Model gelernt werden um daraus die Policy abzuleiten

Modelbased approach: Certainty Equivalence

- Zufällige Aktionen selektieren -> Umgebung erkunden
- Transitions und Rewards beobachten
- T und R statistisch kalkulieren

Modelfreeapproach:

- Lieber Policy statt Model lernen
- zB Qlearning

Was besser ist wird schwer diskutiert

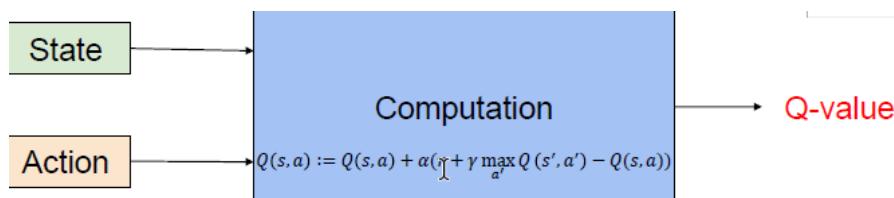
Q Learning

- beliebter RL Algo
- Situation S gegeben
- Erwarteter Reward Q(s,a)

$$\text{Optimal Value } V^*(s) = \max_a Q^*(s, a)$$

$$\rightarrow Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q^*(s', a')$$

- Q Values lernen, indem man a in s performt, man erhält darauf hin reward
- $\rightarrow Q(s, a) := Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
- Explore: Aktionen teilweise random aussuchen
- Exploit: Aktion mit optimalem QValue aussuchen



- Für jede State und Aktion gibt es Feld in Qtable
- Mit Fuzzylearning kann das Lernen verbessert werden: Sammlung von Fuzzyrules ..
- Auch mit Objektfokussiertem Qlearning, Statespace wird als Objektcollection behandelt

Deep Reinforcement Learning

Bisher Situationsset gegeben, aber woher kommt es?

Situation Sets

Situationsvariablen sind oft handgemacht

- Subjektive Featureauswahl
- Nur auf einen Task anwendbar

Situationsvariablen basieren auf Sensorinput

- Man benötigt preprocessing
- Auf mehrere Tasks anwendbar, zB Screenpixel bei Computerspiel

Allgemeiner Ansatz

Versuche Q-Values zu lernen

$$\hat{Q}^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q^*(s', a')$$

$$Q(s, a) := Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Approximator Ansatz

Aktionswert $Q(s, a, \theta) \approx Q^*(s, a)$ schätzen

- Schätzung oft Linear zu 0

Deep Q-Network

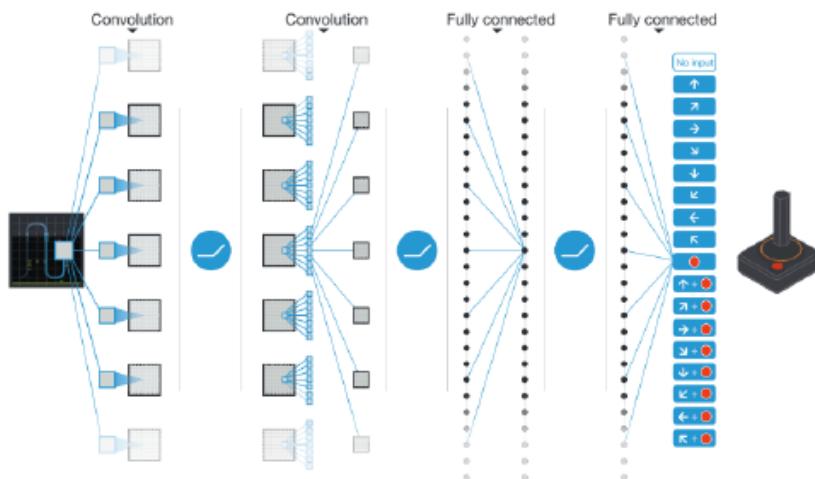
- Nutze DNN für Q^* Schätzung
- Nichtlineare Funktionen in 0
- Annäherung nicht garantiert, gute Generalisierung

Q-Network

- Deep RL Ansatz, Modelfrei, Q-Values mit DNNs lernen
- Label basieren auf vorheriger Data -> no supervised learning

Deep Q Learning

- CNN wird benutzt um Aktionen auszuwählen



- Outperformed die besten anderen Ansätze in vielen Aufgaben

Deep Double Q-Learning

Q Learning schätzt den Greedy Policy Value anhand der momentanen Values

Zweites Gewichtset wird genutzt, um Policy fair zu evaluieren

Stabiler als normales DQL

$$\text{Q-learning } Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q^*(s', a')$$

$$\text{Double Q-learning } Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q'(s', a')$$

Evolving Learning

Next Generation Neural Networks

Spiking NN

- Biologisch realistisches DNN
- Eventbasierter Input
- SNN verrechnet Zeitinformationen abhängig vom Input
- Neuronen mit binärer Aktivierungsfunktion

Wie funktioniert es?

- Oft spärlich verbundenes NN
- Aktivierungsfunktion basiert auf Schwellwerten
- Training basiert auf Spikemining zwischen direkt verbundenen Neuronenpaaren
- Training modifiziert Schwellwert

UseCases

- Pattern recognition, Bild und Audioprocessing, Handschrifterkennung

Vorteile

- Hardware und Energiefreundlich

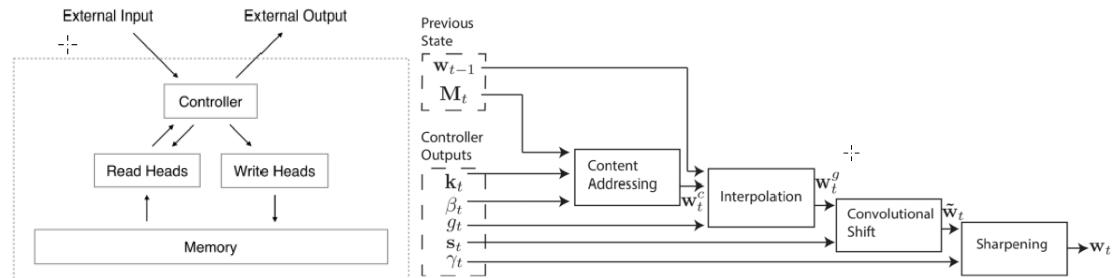
Nachteile

- Gradientenbasierte Optimierungsfunktionen können nicht angewandt werden, weil Aktivierungsfunktionen nicht ableitbar sind
- Ineffiziente Algorithmen führen zu längeren Trainingszeiten

Neural Turing Machines

- NN mit Memorymatrix verbunden, die Attentionmechanismen zum Lesen und Schreiben verwendet
- Löst Aufgaben, die sich lange Sequenzen merken müssen

Wie funktioniert es?



UseCases:

- Sortieren, Kopieren, Recall

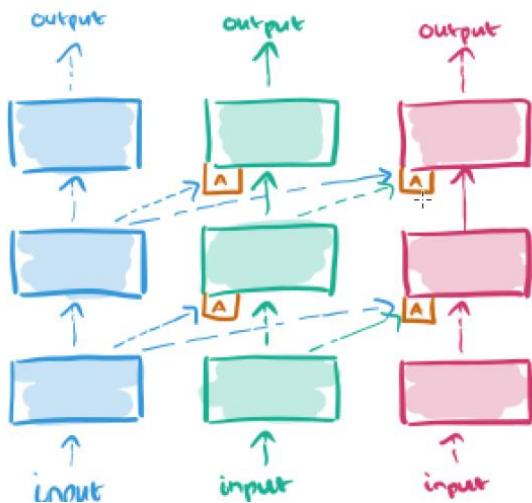
Vorteile sind, dass man für gewisse Probleme weniger Parameter braucht und schreiben/lesen visualisierbar ist

Nachteil ist, dass es nur für ganz bestimmte Aufgaben gut ist

Progressive Neural Networks

Modelling Entscheidungen getrieben von Verlangen nach:

- Lösen von k unabhängigen Aufgaben am Trainingsende,
- Beschleunigung von Lernen durch Transfer
- Vergessen vermeiden



- Eine Aufgabe in blauer Spalte lernen
- Spaltengewichte einfrieren
- Outputs von Layer in Aufgabe 1 wird zusätzlicher Input für Layer+1

UseCases:

- Mehrere Aufgaben in Sequenzen lernen, Transfer ermöglichen, Immun zu vergessen

Vorteile:

- Positiver Transfer

Nachteile:

- Immunität hindert Network daran, gelernte Skills, die auf folgenden Tasks gelernt wurden, für Performanceboost auf vorangegangenen zu nutzen

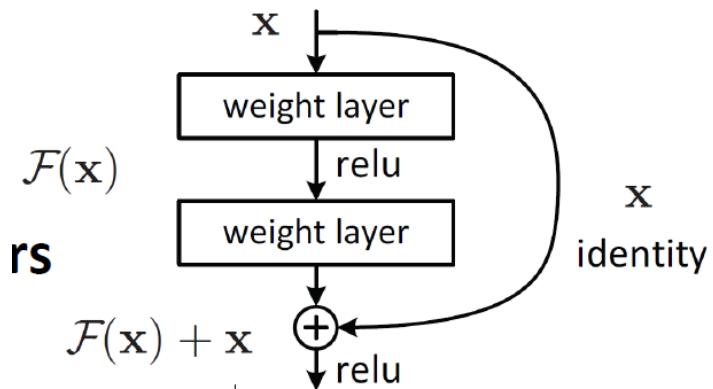
Ziel:

- Use previous Knowledge

Residual Networks

Tiefere NNetze mit Vanishing Gradient und Layer überspringen

Identität aus vorherigen Layern hinzufügen, Weight = 0 heißt Layer wird nichts benutzt

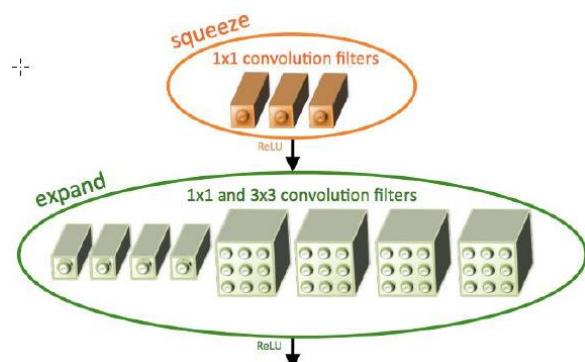


- Für DNN und Image Klassifizierung
- Lernt mit vielen Layern und optimiert sich selbst mit LAYerskip, aber resolved nicht den vanishing Gradient

Squeeze Nets

Parameter reduzieren aber trotzdem Genauigkeit behalten

- 3x3 Filter mit 1x1 Filter ersetzen führt zu 1/9 der Computation
- Anzahl der Inputchannel auf 3x3 Filter reduzieren, indem man 1x1 filter als Bottlenecklayer nutzt
- Spät im Netz downsamplen um große Featuremap zu behalten



- Für Imageklassifizierung und feingranulare Objekterkennung

Vorteile:

- Effizientes, verteiltes Training
- Weniger Memory und bandbreite

Nachteile:

- Keine Garantie dass es für jedes Problem funktioniert

Bayesian Neural Networks

FeedForward Netze in denen Gewichte und Biases mit Verteilung, nicht mit Nummern ausgedrückt werden

- Gewichte werden abgetastet, verschiedene Vorhersagen für mehrere Passes (für einen input)

Wie funktioniert?

- Parameter der Verteilung statt einzelnen skalaren Werten lernen. Wird mit gradientbasierten Optimierern gemacht

UseCases:

- Inputs können mehrere Male durchgereicht werden, um Confidence zu messen
- Hohe Outputvarianz -> Image unbekannt
- Niedrige Outputvarianz -> Als most likely Class klassifiziert

Vorteile:

- Daten identifizieren, die zu keiner Klasse gehören

Nachteile:

- Man muss mehrere Durchläufe machen

Transformer Model

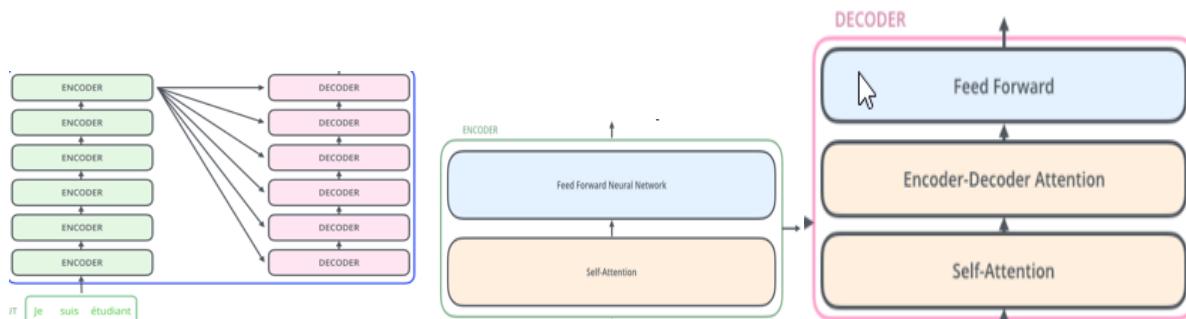
Für S2S Learning ohne RNN

Idee:

- RNN kann Input nicht parallel verarbeiten, erhöhte Trainingszeit
- Transformer als Kombi aus CNN und Attention Models

Wie?

- Encoder/Decoder Setup: Intern ähnliches Setup wie RNN s2s Model
- En: Jedes Enc. Layer hat self attention Mechanismus, welches die Wichtigkeit von Input einfängt, und FFN
- DeCoder besteht aus EnDe Attention Mech und FFN, das hilft relevante Parts der Input sequenz zu fokussieren



UseCases:

- NLP: Übersetzen, Sprache erkennen, TextToSpeech

Vorteile:

- Mehr Training da Parallel

Nachteile:

- Keine Sequenzielles Processing, Modelkomplexität erhöht

Efficient Inference Algorithms

Dass DL Models genauer werden ist mit Kosten verbunden: Größere Models, Trainingszeiten, weniger Energieeffizienz.

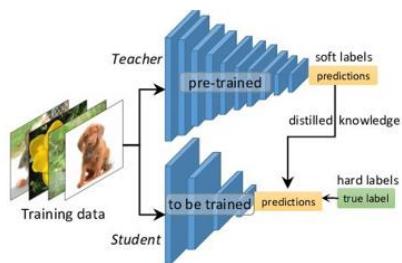
Für Mobile Devices müssen Networks in die andere Richtung gehen.

- Pruning kilt bis zu 90% der Neuronen ohne signifikante Performanz zu verlieren
 - Idee: Neuronen bewerten nach Contribution, niedrige entfernen
-
- Quantization: Anzahl Bits reduzieren, die Nummer repräsentieren
 - Gewichte und Aktivierung kann von 8bit Integer repräsentiert werden, ohne Genauigkeit zu verlieren

Teacher Student Networks

Trainingsprozess zum produzieren kleinerer Netzwerke

- Mithilfe großem, vtrainierten Netz ein kleines Netz trainieren
- Anhand Output Featuremaps des großen Netz, verhalten nachahmen



1. Trainernetz trainieren
2. Welche Layer braucht das kleine?
3. Daten durch Trainernetz um benötigte Outputs zu bekommen
4. Backprop durch kleines Netz mithilfe der Outputs aus 3

UseCases: Mit starker Hardware trainieren, mit schwacher Benutzen

Vorteil: Overfittet nicht so leicht: weniger Komplexität aber trotzdem Performance

Nachteil: Trainingszeit, keine Garantie über gleiches Wissen

Lottery Ticket Hypothesis

Trainingsprozess zum produzieren kleinerer Netzwerke

- Parameteranzahl verringern
 - Ein großes Netzwerk enthält kleineres Subnetz
1. NN Random initialisieren
 2. Trainieren
 3. Einen Teil prunen
 4. Gewichte des geprunten auf die Werte von 1 resetten
 5. Das geprunte erneut trainieren

UseCases: Für Devices mit Bottleneck, für devices mit Healthsettings, da Übertragung sensitiver Daten minimiert

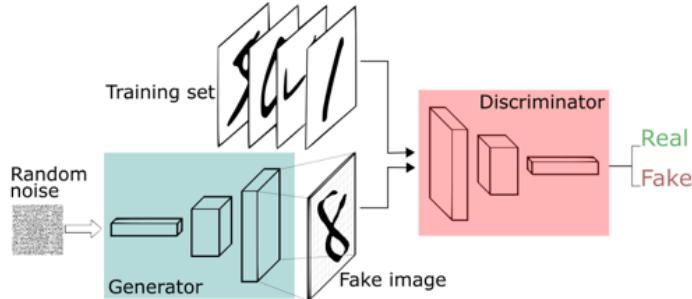
Vorteil: Geringerer Storage und Interferenzzeit, kann in stärker performendem Netz resultieren

Nachteil: Sehr langes Training, keine Garantie auf „winningticket“ finden

Generative Adversarial Networks

Kann neuen Content für Datenvermehrung produzieren

- Besteht aus Generator: versucht Diskriminatior zu verarschen indem er realistische Beispiele produziert
- und Diskriminatior: lernt, zwischen echt und falsch zu unterscheiden
- Spiel endet wenn Diskriminatior es nicht schafft



- Beide haben Lossfunctions
- Trainiert mit alternating GD: Erst Parameter für G fixen und GD Iteration bei D mit echten und generierten Bildern machen.
- Dann D fixen und G für einzelne Iteration trainieren

Vorteil:

- Realistische Samples generieren

Nachteil:

- Non-convergence_ Model parameter nähern sich nie an, instabil
- Model collapse: Generator kollabiert -> limitierte Varianz bei Samples, irgendwann alle gleich
- Diminishing Gradient: Discriminator wird so gut, dass der Generator gradient verschwindet und nichts lernt

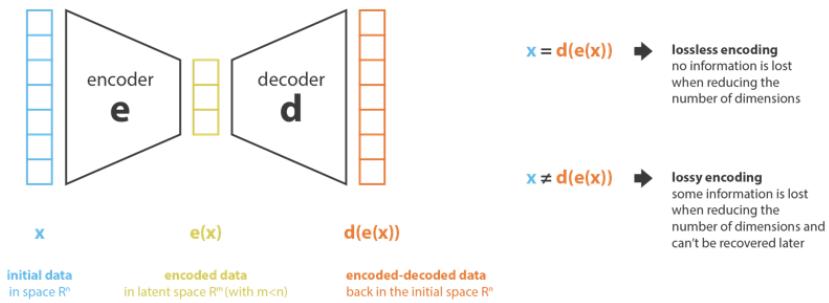
Variational Autoencoders

Data generieren

- Mit Autoencoderstruktur
- Wahrscheinlichkeitsverteilung lernen
- Es werden samples von dieser Verteilung genommen und durch Decoder gepusht

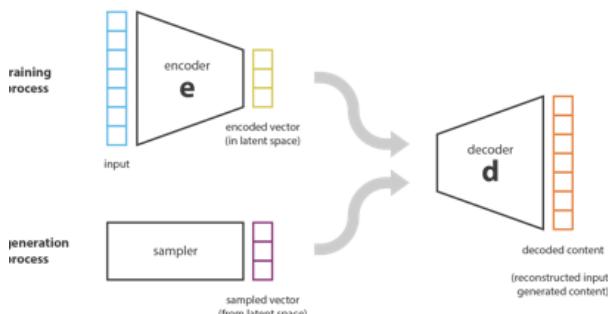
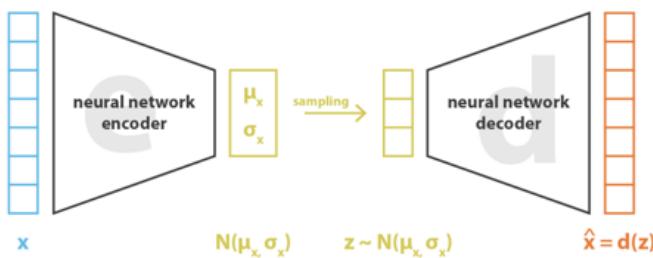
Was ist Autoencoder?

- Autoencoder Ist paar aus verbundenen Netzwerken, einem encoder und einem decoder
- Encoder Netz nimmt Input und konvertiert in kleinere Repräsentation, auch latent space genannt
- Decoder converts latent zurück zu original Input



Was ist Variational Autoencoder?

- Um Dten zu kreieren Constrain auf Encodingnetz legen, das es zwingt latent Vektoren zu generieren, die grob unit guassian distribution folgen
- Aus dieser learnt distribution werden samples genommen und druch decoder gepusht, so entsteht data
- Während Training lernt Netz tradeoff zwischen Genauigkeit und wie sehr die Latentvariablen der Unit Guassian Distribution ähneln



Training

1. Input als Distribution über Latentspace codieren
2. Punkt von Latentspace als Probe aus dieser Distribution nehmen
3. Dieser Punkt wird kodiert und reconstruction error berechnet
4. Dieser Error Backprop durch das Netz

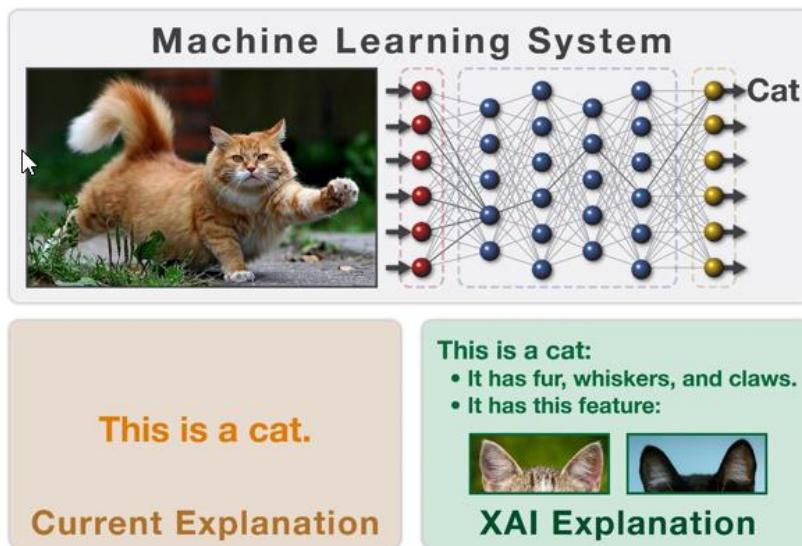
Vorteil

- Behandelt GANS Schwächen

Nachteil

- Nicht garantiert, dass latent variables einer Gaussischen Verteilung folgen, weil man evtl aus unangemessener Verteilung sampled

Explainable AI



- ML kann von Bias und Shortcuts beim lernen beeinflusst werden: Selection Bias, Modelling Bias
- Deshalb brauchen wir Explainability: Das Innere von MLSystems verstehen, wie und warum wurde Entscheidung gefällt?
- GDPR in EU bestimmt, dass Patienten wissen dürfen wie Entscheidungen getroffen werden
- Internal Knowledge sollte auch dabei helfen, Genauigkeit und Generalisierung zu verbessern

Definitions

- Interpretable ML: Methoden und Modelle, die das ML Verhalten verstehtbar machen
- BlackBoxSystems: Modelle, die nicht interpretierbar und verstehtbar anhand der Parameter sind
- Interpretability: Grad des Verstehens, Grad zu dem ein Mensch das Modelergebnis vorhersagen kann
- EXPLANATION ist Antwort auf „warum“ Frage

Importance of Interpretability

- **Unvollständige Beschreibungen**
- **Unvollständige Problemformalisierung:** manchmal löst Vorhersage nur teil des Startproblems, hier braucht man Erklärung
- **Gain Knowledge:** Ziel der Wissenschaft ist Wissen, aber Blackbox bunkert Wissen anstatt es in Data zu spiegeln
- **Gesundheit und Sicherheit:** Bei Risikotasks z.B. im Verkehr muss man wissen, was die Maschine denkt
- **Biases:** Detection
- **Debug und Audit:** Fairness, Privacy, Robustness, Causality, Trust
- Nicht so wichtig, wenn das System keinen realworld impact hat, das Problem wellstudied ist
- Interpr. könnte Leute dazu befähigen das System zu manipulieren

Taxonomy of Interpretability Methods

Intrinsic oder post hoc?

- Interpretability, weil die Komplexität eingeschränkt wird (intrinsic) oder weil das Model nach Training analysiert wird?
- Int: ML Models, die wegen simpler Struktur interpretierbar sind
- PH: Interpretation methods

Models anhand Ergebnisse differenzieren

- Feature Summary Statistic
- Feature Summary Visualization
- Model Internals: zB learned Weights
- Data Point: Methods that return data to make model interpretable
- Intrinsically interpretable Model

Modelspezifisch oder agnostisch?

- Ms: limitiert auf bestimmte Modelklassen
- A: Egal welches Model, nach Training angewandt

Local oder global?

- Individuelle Vorhersage oder komplettes Modelverhalten?

Interpretable Models

Interpretability, indem Komplexität eingeschränkt wird

Linear Regression

- Vorteile: Entscheidungen transparent, Gewichtanzahl regulieren, einfach
- Nachteile: Nur für lineare Beziehung, poor predictive Power

Logistic Regression

- Vorteile: gibt Wahrscheinlichkeit zurück, leicht lernbar
- Nachteile: Poor predictive power, komplett Abspaltung. Wenn ein Feature 2 Klassen perfekt trennen kann, kann model nicht trainiert werden

Decision Trees

Spalte die Daten mehrfach anhand bestimmter Werte in Features

- Vorteile: Interactions between features, visualization mit nodes und edges, gute Erklärungen
- Nachteil: nicht gut bei linear, unstabil wenn Data geändert wird(dann ändert sich Tree stark), schlechte klassifizierung

Model-Agnostic Methods

- Erklärung von Model trennen
- Flexibel, bessere Performance da komplexe Models genutzt werden können, leichter vergleichbar mit anderem Modell
- System ist nicht an bestimmte Erklärungsform gebunden

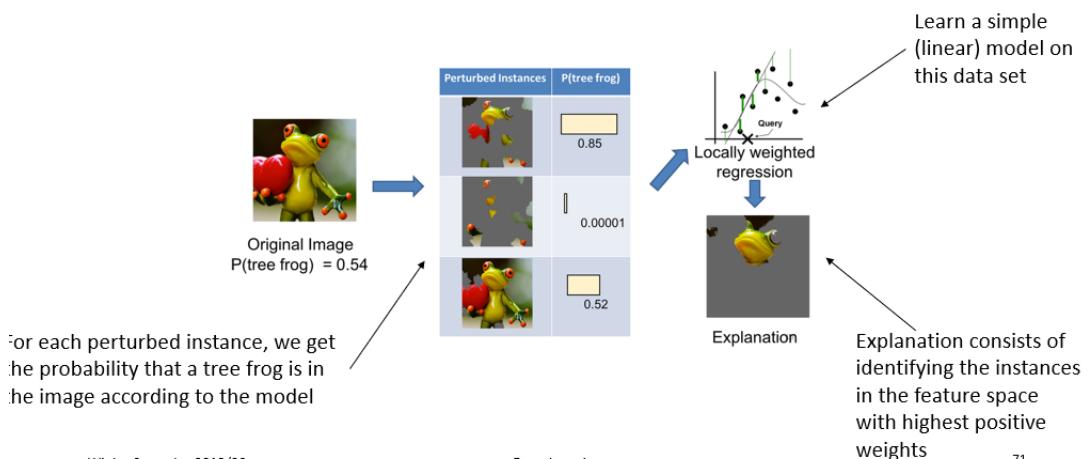
- Repräsentation der Methode flexibel

Mehrere Ansätze

- Partial Dependence Plots (Relationship zwischen Input und Prediction)
- Individual Conditional Expectation Plots (individuelle Unterschiede und Gruppenunterschiede zwischen modelinputs finden)
- Accumulated Local Effects Plots (wie beeinflussen Features die Vorhersage im Schnitt)
-
-
- Local Interpretable Model-Agnostic Explanations (LIME)
- Trainiert lokale Modelle die Vorhersagen der Blackbox zu schätzen
- LIME testet wie sich Datavariation auf Vorhersage auswirkt
- Generiert neues Dataset und entsprechende Vorhersage der Blackbox
- Lime nutzt dieses Dataset um interpretierbares Model zu trainieren

Lime Key steps:

1. Trainiert blackbox
2. Wähle Instanz aus die du erklären willst
3. Störe dein Dataset und bekomme Vorhersagen für diese neuen Punkte
4. Trainiere gewichtetes, interpretierbares Model auf den Datenset mit Variationen (gewichtete basieren auf neue Samples, anhängig von der Nähe der Störkraft instanz zum original)
5. Erkläre die Vorhersage indem du lokal mode interpretierst



Vorteile:

- Man kann unterliegendes MLModel austauschen und trotzdem dasselbe lokale, interpretierbare Model für Erklärungen nehmen
- Einfache lokale Modelle für Erklärung benutzen
- Leicht, Python und R

Nachteile:

- Gaussche sampling benutzt, um Datainstanz zu stören
- Potenzielle instabile Erklärungen
- Kein globaler Approach, könnte unzureichend sein in manchen Szenarien

Example-based Explanations

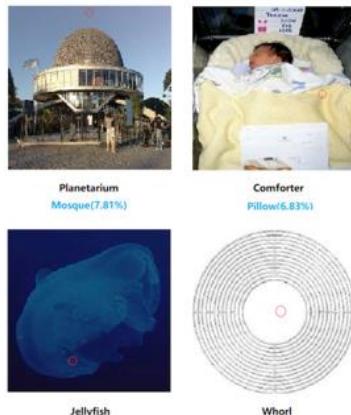
- Bestimmte instanzen des Datasets auswählen um Verhalten zu erklären
- Auch für unterliegende Datenverteilung nutzen
- Unterschied zu agnostic ist dass examplebased Datainstanzen auswählt, nicht features zusammenfasst
- Macht nur Sinn wenn man die Dateninstanz in verstehtbarer Form repräsentieren kann

Ansätze:

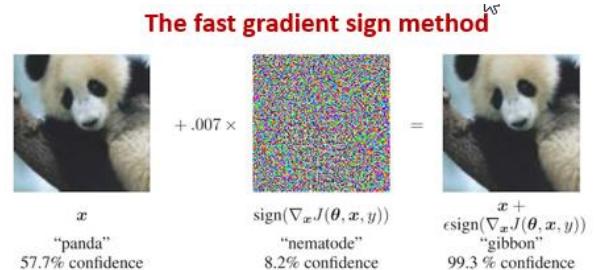
- COUNTERFACTUAL EXPLANATION (wie muss Instanz ändern um Vorhersage zu ändern)
- Adversarial example
Instanz mit kleinen, absichtlichen Featurestörungen die MLModel zu falscher Vorhersage zwingen
- Wichtig, da anfällig für Attacken

• Adversarial Methods

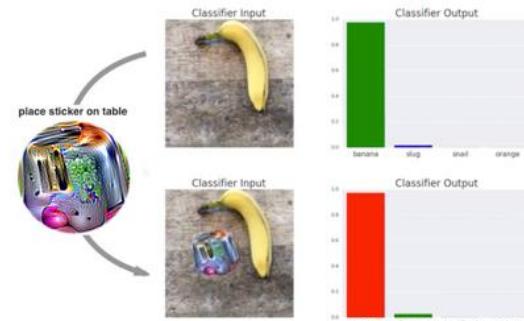
One pixel Attack



Winter Semester 2019/20



The Adversarial patch



- Prototypes and Criticisms (Auswahl von giuter oder schlechter repräsentativer Instanz)
- Influential INSTANCES (Datenpunkte trainieren die am meisten Einfluss auf die Parameter eines Model hatten)
- K-neares neighbour (Ein interpretierbares ML modell basierend auf Beispielen)

Neural Network Interpretation

- Eine einzelne Vorhersage bei DL Systemen kann Millionen mathematischer Operationen enthalten
- Interpretation ist unmöglich
- Standarderklärmethoden können funktionen aber NN spezifische Ansätze sind vorteilhaft
- ➔ Nutze gelernte Information in hidden layers und den Gradienten

Feature Visualisation

- Hauptvorteil von DDN ist dass die automatisch highlevelfeatures in Hidden Layers lernen
- Erstes Layer sowas wie ecken und simple texturen
- Späteres Layer mehr komplex: Texturen und Strukturen
- Letzte Layer können Objekte oder Objektparts

- Input finden dass die Unitactivation maximiert
- Feature visualisierung ist optimierungs Problem:
Anstatt activation maximieren, kann man auch minimieren
- Keysteps:
 1. Random noise
 2. Constraints auf Update, nur kleine Änderungen
 3. Jittering Rotation der Scaling um noise zu reduzieren

Vorteile:

- Einzigartige Einsichten in learning
- Welche pixel waren wichtig für Klassifizierung?

NACHTEIL:

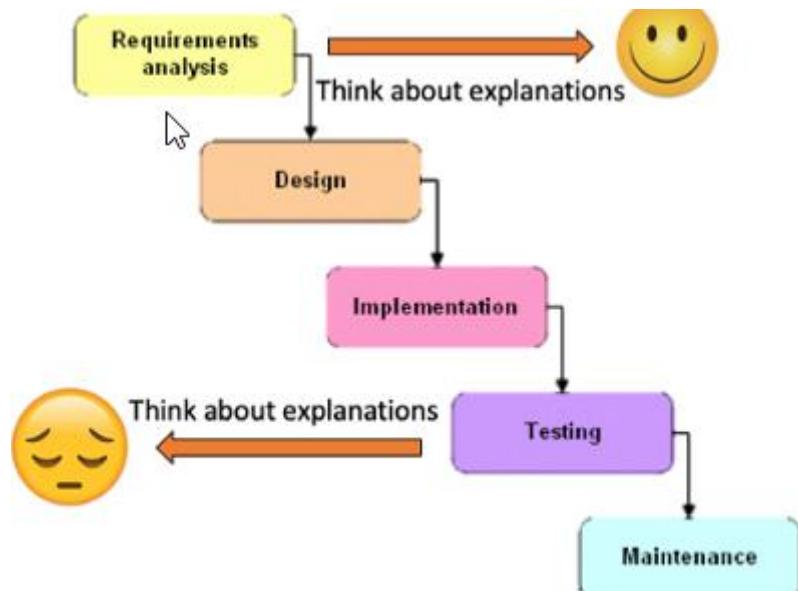
- Viele Visualisierungsbilder nicht interpretierbar
- Illusion der Erklärbarkeit , kein wirkliches Einsehen,

Where to next?

- Früh über Interpretierbarkeit nachdenken

Momentanes Dilemma: Wir brauchen komplexe nichtlineare Blackbox für Probleme, einfache Models(erklärbar) führen zu schwächerer Performance

Lösung: Explainability in AI design integrieren



Wrap

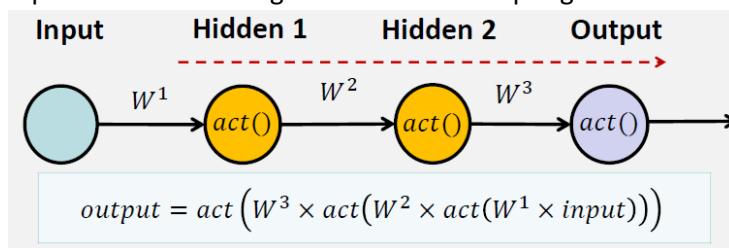
Int

FFN

- Errorsquare, weil große Errors größer und kleine kleiner werden
- Außerdem will man Positive Errors wegen auscanceln

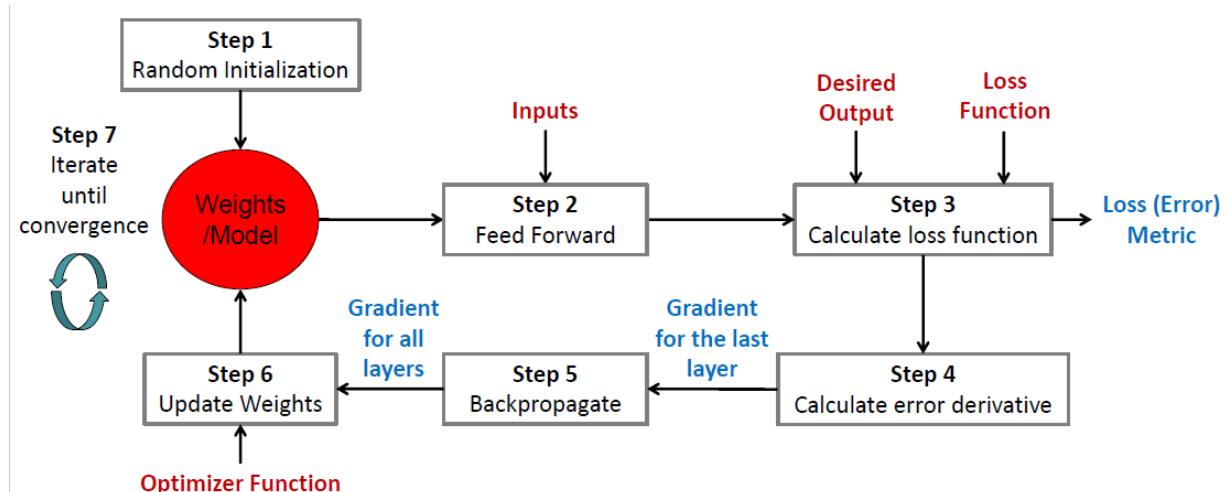
```
pred = input * weight  
  
error = (pred - goal_pred) ** 2  
  
derivative = input * (pred - goal_pred)  
  
weight = weight - (alpha * derivative)
```

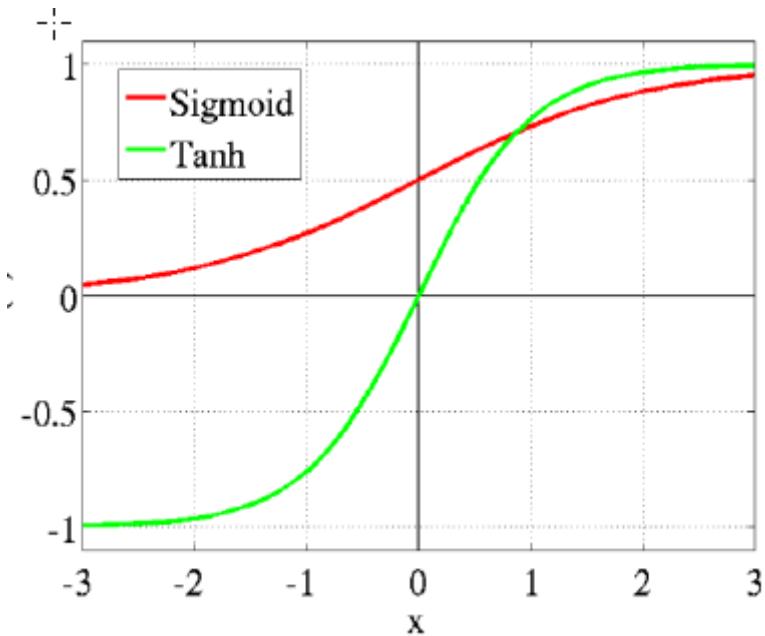
- Derivative bestimmt Richtung und Größe
- Alpha minimiert Divergenzeffekte wenn Input groß ist



- Regression (SingleNumValue): Linear, MSE
- Binary Outcome (is or isn't a class): Sigmoid, Binary Cross Entropy
- Single LabelMulticlass: Softmax, CrossEntropy
- Multiple labels Multiclass: Sigmoid, Binary Cross Entropy

- Optimiser passt die Gewichte mithilfe des Ergebnisses der Lossfunction an
- Gradient Descent, langsam
- Momentum, beschleunigt Konvergenz mithilfe des letzten Gradient
- NAG, berechnet Gradient mit nächstem Step
- AdagradAGA, Adaptiv
- Adadelta, Adaptiv, schränkt Fesntergröße vergangener Gradienten ein
- ADAM AME, first second moment





$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad \sigma(x) = (1 + e^{-x})^{-1}$$

Sig:

CNN

- Verschiedene Patterns in verschiedenen Regionen, Downsamplen ist egal, manche sind viel kleiner als ganzes Bild
- Convolution -> Max Pooling -> Convolution -> Max Pooling -> Flatten -> FFN -> Prediction
- Convolutional Layer wenden Filter aufs Bild an, Filter haben gewichtete, Berechnen von Dotprodukt zwischen Filter und kleinem Chunk des Bildes, für alle Chunks
- Was von Convolution ausgespuckt wird wird downsampled (MaxPooling) zB 4 Pixel auf 1 reduzieren
- Kleine Inputveränderungen machen sich nicht in MaxPooling output bemerkbar
- Kann man lange wiederholen um so Strukturen runterzubrechen

RNN

- Weil man Sequenz liest muss man ganzen Output merken und nicht nur einzelne Str#nge deshalb braucht man Memory
- Output of Hidden Layer in Memory gespeichert, als zusätzlicher Input für nächsten Step

$$h_t = f_w(h_{t-1}, x_t)$$

↓
 current state
 ↓
 state in memory
 (previous state)
 ↓
 current input vector

←
 function with
 parameters W

- Unrolled RNN nutzt dieselbe Weightmatrix an jedem Zeitschritt
- Weil so viele Matrixmultiplikationen mehrfach gemacht werden, kann gradient v oder e

- RNN leiden an Kurzzeitgedächtnis, können sich lange Sequenzen nicht merken

Gated RNN

- Gated RNNs kreieren Pfade deren Ableitungen nie vanishen oder exploden
- Gated RNN können alten State vergessen und lernen selbst wann das zu tun ist

LSTM

- LSTM: Informationen vergessen oder behalten
Cell State überträgt Information über den ganzen Prozess der Sequenverarbeitung
ForgetGate entscheidet was man von letztem Step behält
InputGate entscheidet was man von aktuellem Step addiert
OutputGate entscheidet was der nächste Hiddenstate sein soll
- Gates lernen Relevanz während Training
- LSTM vergisst nie alte Information, wenn ForgetGate nicht geschlossen wird, deshalb kein Vanishing Gradient

GRU

- GRU
Reset: Welche alte Info vergessen
Output: Zusammenaddieren von Info, in nächsten Step einspeisen
Update: Welche Alte Info weg und welche neue Dazu

S2S

- Input Sequenz lesen und ausgeben, zB Übersetzung
- zB Zwei RNNs in EncoderDecoder Archi
- Encoder wandelt Sequenz in Vektor um, Decoder benutzt weiteres RNN um Encoderinput zu Output zu machen
- Nur der letzte Part des Encoders wird benutzt um Decoder zu initialisieren, wenn die Sequenz lang ist wird dieser einzelne Vektor dann so lang dass dies zum Bottleneck wird

Attention Mec

- Deshalb Attentionmechanisms um Kontextvektor zu bauen, er wird beim Decoden benutzt
- Er sammelt globale Information
- Unterschiedlicher Fokus auf unterschiedliche Inputparts, jeder Input bekommt Score
- Aus Encoderhidden States wird Contextvektor produziert
 1. Scores von States nehmen
 2. Alle durch Softmax Layer
 3. Jeden State mit softmax score multi
 4. Vektoren summieren
 5. Decoder mit Ergebnis füttern

Regularisation

- Opfert Komplexität für weniger Overfit
- Lossfunction wird Term hinzugefügt, der hohe Gewichte bestraft, weil diese das Netzwerk instabil machen
- Ziel ist robuste Predictionfunction
- Tradeoff zwischen Bias und Varianzfehler
- Modelkomplexität steigt, weniger gen. Error weil Modelbias sinkt
- Steigt sie noch weiter, steigt gen. Error wieder weil Modelvarianz steigt

L1

- Weight Penalty: Summe absoluter Gewichtvalues

L2

- Weight Penalty: Summe quadrierter Gewichtvalues

Data Augmentation

- Bessere Generalisierung durch mehr Data
- Fakedata

Training with Noise

- DAmit sich Netzwerk Trainingsbeispiele nicht merken kann, weil Input immer changed
- Kleinere gewichte und niedrigerer gen Error

Dropout

- Mehrere gleiche Netzwerke werden zusammengeschmolzen, weil alle unterschiedliche Fehler machen aber zusammen sind sie stark. Einzelne Neuronen werden rausgeworfen dann geschmolzen

Early Stopping

- Rechtzeitig stoppen bevor Overfitting
- Wenn Gen Error steigt stoppen

Compression

Pruning:

- Neuronen nach Beitrag bewerten, schlechte raus schneiden
- Kleineres und schnelleres Netzwerk

Quantization

- Bits von Weights und Aktivierungen reduzieren von 32 auf 8, ohne großartig Genauigkeit einzubüßen

Minibatchlearning

- Performt update für kleinen Batch n an Trainingsbeispielen
- Reduziert Noise in Gewichtupdate Varianz
- Vorteile sind Stabile Konvergenz und guter Lernspeed, gutes Lokales Minima
- Beachtet nicht kompletten Loss da Data gesplittet

Explainable AI

LIME

- Trainiert Lokales Model darauf, die Vorhersagen einer unterliegenden Blackbox zu schätzen

Feature Visualisation

Handouts und Homework

2

Gradient einfach jede Variable Ableiten und als Vektor schreiben

3

Forwardpropagation durch ein Layer

```
def fcc_one_layer(X, W, b, activation):  
    return activation(np.matmul(X, W) + b)
```

2 Hidden layer:

```
class fcc:  
    def __init__(self, n_input, n_hidden1, n_hidden2, n_out):  
        #parameters  
        self.W_i_h1 = np.random.randn(n_input, n_hidden1)  
        self.b_h1 = np.random.randn(n_hidden1)  
        self.W_h1_h2 = np.random.randn(n_hidden1, n_hidden2)  
        self.b_h2 = np.random.randn(n_hidden2)  
        self.W_h2_o = np.random.randn(n_hidden2, n_out)  
        self.b_out = np.random.randn(n_out)  
        #neuron activations and input H^n  
        self.X = None  
        self.h1 = None  
        self.h2 = None  
        self.out = None  
        #components of the gradient  
        self.dW_i_h1 = None  
        self.db_h1 = None  
        self.dW_h1_h2 = None  
        self.db_h2 = None  
        self.dW_h2_o = None  
        self.db_out = None  
  
    sigmoid(X):  
        return 1/(1 +np.exp(-X))  
  
    softmax(X):  
        #more stable  
        eps = X.max()  
        return np.exp(X + eps)/(np.sum(np.exp(X + eps), axis=1).reshape((X.shape[0], 1)))
```

```
def forward_propagation(self, X):
    self.X = X
    # (3.4)
    self.h1 = fcc_one_layer(X, self.W_i_h1, self.b_h1, sigmoid)
    # (3.4)
    self.h2 = fcc_one_layer(self.h1, self.W_h1_h2, self.b_h2, sigmoid)
    # (3.4)
    self.out = fcc_one_layer(self.h2, self.W_h2_o, self.b_out, softmax)
    return self.out
```

4

5

6