Neuronale Netze

Zusammenfassung

Marius Bauer

1. August 2017

Inhaltsverzeichnis

1 Neuronale Netze - kurzer Überblick

Wieso werden neuronale Netze verwendet?

- Massive parallelism.
- Massive constraint satisfaction for ill-defined input.
- Simple computing units.
- Many processing units, many interconnections.
- Uniformity (-> sensor fusion)
- Non-linear classifiers/ mapping (-> good performance)
- Learning/ adapting
- Brain like ??

Wofür werden neuronale Netze verwendet?

- Classification
- Prediction
- Function Approximation
- Continuous Mapping
- Pattern Completion
- Coding

Welche Kriterien müssen beim Entwurf von neuronalen Netzen beachtet werden?

- Recognition Error Rate
- Training Time
- Recognition Time
- Memory Requirements
- Training Complexity
- Ease of Implementation
- Ease of Adaptation

Welche Parameter werden typsicherweise vom Entwerfer bestimmt?

- Net Topology
- Node Characteristics
- Learning Rule
- Objective Function
- (Initial) Weights
- Learning Parameters

Wo werden neuronale Netze verwendet?

- Space Robot*
- Autonomous Navigation*
- Speech Recognition and Understanding*
- Natural Language
- Processing*
- Music*
- Gesture Recognition
- Lip Reading
- Face Recognition
- Household Robots
- Signal Processing
- Banking, Bond Rating
- Sonar

Advanced Neural Models

- Time-Delay Neural Networks (Waibel)
- Recurrent Nets (Elman, Jordan, Robinson,..)
- Higher Order Nets

- Modular System Construction
- Adaptive Architectures
- Hybrid Neural/Non-Neural Architectures
- Deep Nets

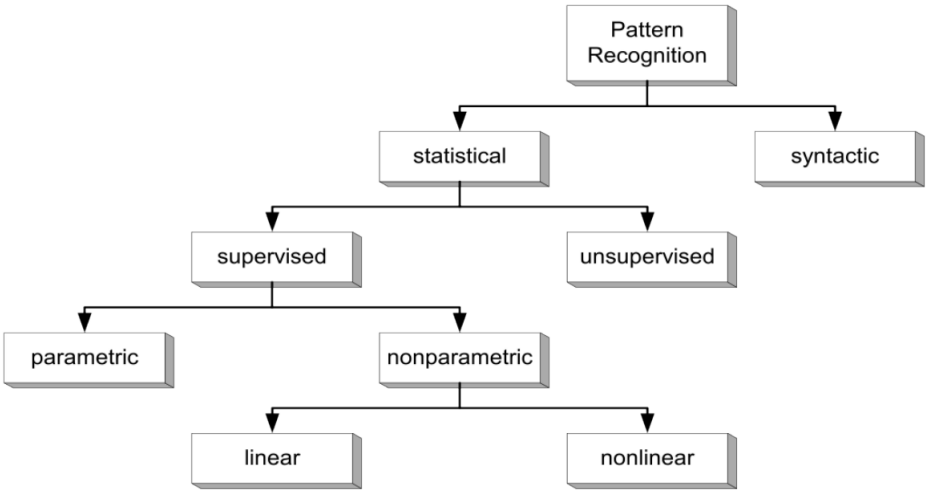
Welche Probleme können beim Entwurf von neuronalen Netzen auftreten?

- Local Minima
- Speed of Learning
- Architecture must be selected
- Choice of Feature Representation
- Scaling
- Systems, Modularity
- Treatment of Temporal Features and Sequences

Eigenschaften von neuronalen Netzen:

- non-linear classifier
- approximate posterior probability
- non-parametric training

2 Pattern Recognition



- supervised:
  - labels / classes of training data are known
  - train a classifier
- unsupervised:
  - labels / classes are not known
  - find the structure of the data
  - methods: clustering, autoassociative nets
- parametric:
  - assume underlying probability distribution
  - estimate the parameters
  - eg.: Gaussian classifier
- unparametric:
  - doesn't assume underlaying probability distribution
  - estimate probability of error from training data
  - nearest neighbours, parzen window, perceptron

2.1 Bayes Decision Theory - Parametric

$$P(\omega_j|x) = \frac{p(x|\omega_j) \cdot P(\omega_j)}{p(x)}$$
$$p(x) = \sum_j p(x|\omega_j) \cdot P(w_j)$$

Since we have overlapping probability distribution false classifications occur. Simply speaking to reduce false classification always decide for the class with the highest probability (errors still can occur!).

2.2 Classifier Discrimination Function

The *classifier discrimination function* iterats over all classes and choses those which has the highest probability. In the case of the Gaussian classifier it looks as follows:

$$\begin{aligned} g_i &= P(\omega_i|x) \\ &= p(x|\omega_i) \cdot P(\omega_i) \\ &= \log(p(x|\omega_i)) + \log(P(\omega_i)) \end{aligned}$$

Problems:

- limited traning data
- limited computation
- class-labeling potentially costful and errorful
- classes or features might not be known

Parametric solution: assume that $p(x|\omega_i)$ has a parametric form. The most common representative: multivariant normal density.
Univariant normal density:

$$p(x) = \frac{1}{\sqrt{2 \cdot \pi\sigma}} \cdot e^{(-\frac{1}{2} \cdot \frac{x-mu}{\sigma})^2}$$

Multivarian normal density:

$$p(\overleftarrow{x}) = \frac{1}{\sqrt[d]{2\pi} \cdot \sqrt{|\Sigma|}} \cdot e^{-\frac{1}{2} \cdot \frac{(\overleftarrow{x} - \overleftarrow{\mu})^2}{\Sigma}}$$

The following form takes effect for the classifier discrimination function:

$$g_i(\overleftarrow{x}) = -\frac{1}{2} \cdot \frac{(\overleftarrow{x} - \overleftarrow{\mu_i})^2}{\Sigma_i} - \frac{d}{2} \cdot \log(2\pi) - \frac{1}{2} \cdot \log(|\Sigma_i|) + \log(P(\omega_i))$$

For the Gaussian classifier we have to estimate the *covariance matrix* Σ_i and the *mean vector* μ_i. To estimate the parameters we can use the *MLE: maximum likelihood estimation* for the univariant case. For the multivariant case we can use:

$$\overleftarrow{\mu} = \frac{1}{N} \cdot \Sigma_{k=1}^{N} \overleftarrow{x_k}$$

$$\Sigma = \frac{1}{N} \cdot \Sigma_{k=1}^{N} (\overleftarrow{x_k} - \overleftarrow{\mu}) \cdot (\overleftarrow{x_k} - \overleftarrow{\mu})^T$$

2.3 Curse of Dimensionality

The problem with the classifier design in general is that we can not say which features are the most valuable ones, are more features better and is features are useful will they be ignored?
However in general we can say that more features performe worse because we have limited training data and we still have to estimate the parameters (Eg. 1000 sample training data and 1000 parameters to estimate). We have to select the best feature and might have to reduce the dimensions (for example with the *Principle Component Analysis (PCA)*.

2.3.1 Principle Component Analysis (PCA)

The idea is that single dimension are correlated and we want to remove the dimension with the least informations:

1. find the axis with the highest variance
2. rotate the space along the axis
3. not the dimensions are uncorrelated
4. remove the dimension with the lower variance

2.4 Non-Parametric Methods

Non-Parametric Methods do not assume any distributions and try to find structures in the data itself.

2.4.1 Parzen Window

1. chose a window with volume V
2. count the numbers of samples in that window $p(x) = \frac{\frac{k}{N}}{V}$ where k are the samples in the window and N is the total numbers of samples.

The hard part is to chose the window size. the thumb roule is: $V_n = \frac{1}{\sqrt{n}}$

2.4.2 k-nearest Neighours

3 Recurrent Neural Networks

Recurrent Neural Networks unterscheiden sich von anderen neuronalen Netzen in dem Punkt, dass sie einen (oder mehrere) Zustände speichern und verwenden können. Dabei ist ein Zustand ein berechneter Wert des vorherigen Zeitschritts. Diese Zustände können widerrum als Input genutzt werden. Somit ist eine zeitliche Zustandsspeicherung möglich.

3.1 Sequence Learning

Sequenzen sind überall in unserem alltäglichem Leben vertreten: z.B. bei der Sequenzierung von Töne in Sprache. Bei der Spracherkennung wird der Fokus auf Wortsequenzen gelegt. Bei sequence learning gibt es vier generelle Probleme:

- sequence prediction: Was wird das nächste Element in der Sequenz sein?
- sequence generation: generieren einer Sequenz (eg. Wortsequenz)
- sequence recognition: Ist die Sequenz legitim?
- sequence decision making: ??

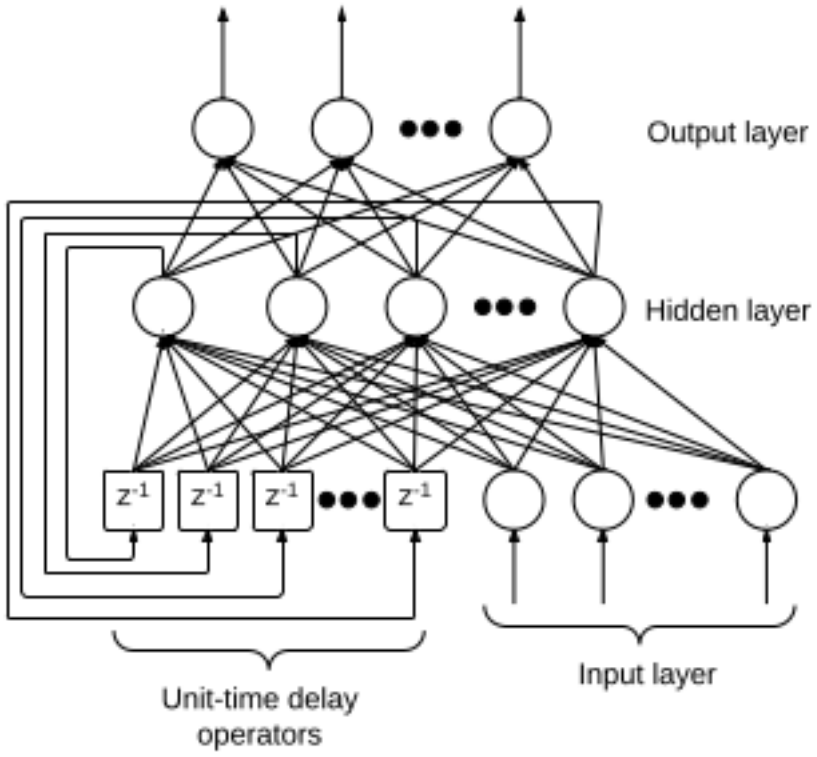
3.2 Elman vs. Jordan Networks - Simple RNN

Der Unterschied zwischen *Elman* und *Jordan* Netzen besteht darin, dass sie unterschiedliche Outputs zwischenspeichern und wiederverwenden. Bei Elman Netzen wird der Output des Hidden Layers als Input für den nächsten Zeitschritt verwendet. Jorden Netze auf der anderen Seite nutzen den Output des Netzes als Input für den nächsten Zeitschritt.

3.3 Aufbau

- Input Units $X = x_1, x_2, ..., x_n$
- Output Units $Y = y_1, y_2, ..., y_n$
- Hidden Units $H = h_1, h_2, ..., h_n$
- Connections between Units

Im Prinzip sehen RNN genauso aus wie NN. Jedoch sind die Verbindungen anders. Diese können nämlich zurückführen und als Input dienen. Somit lassen sich vergangene States wiederverwenden. Alternative könnte man *recurrent hidden units* in den Aufbau mit aufnehmen. Diese speichern vergangene Zustände und leiten diese als Input in den darauffolgenden Zeitschritt in die entsprechenden Units.

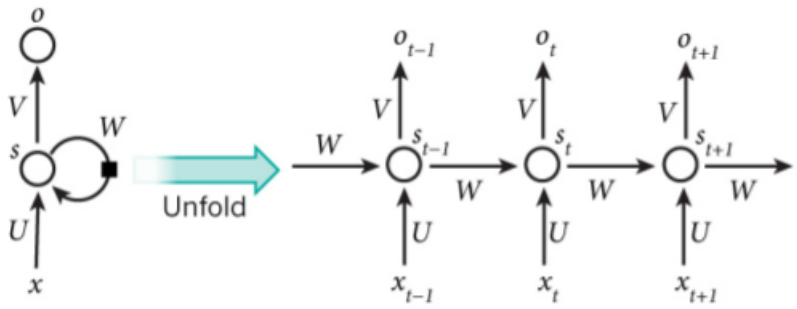


3.4 Training

RNN werden mittels *backpropagation through time* trainiert. Dies ist eine Erweiterung des ursprünglichen Backpropagation, bei der die wiederkehrenden Zustände berücksichtigt werden. Dazu wird ein RNN *ausgerollt* um den zeitlichen Zusammenhang darzutellen: Zu jedem Zeitpunkt nutzen wir *forward pass* um die Aktivität in jedem Zeitschritt zu berechnen. Mittels *backward pass* wird die Fehlerableitung zu jedem Zeitschritt berechnet. In dem entrollten Netzwerk kommen die selben Gewichte öfters vor (shared weights), zu sehen in Bild XX. Jede Instanz eines Gewichtes muss den selben Gradienten erhalten (Einschränkung):

1. $w_j^{t_1} = w_j^{t_2} \Rightarrow \Delta w_j^{t_1} = \Delta w_j^{t_2}$
2. Berechne $\frac{\partial E}{\partial w_j^{t_1}}$ und $\frac{\partial E}{\partial w_j^{t_2}}$
3. $\Delta w_j^{t_1} = \Delta w_j^{t_2} = -\eta \left(\frac{\partial E}{\partial w_j^{t_1}} + \frac{\partial E}{\partial w_j^{t_2}} \right)$

Hierbei kann zum einen die Summe betrachtet werden, aber auch der Durchschnitt!



3.5 Vanishing / Exploding Gradient

4 Speech

Die Sprachproduktion beim menschlichen Körper besteht aus drei Teilen: 1. dem supraglottale Sprachtrakt, 2. dem kehlkopf und 3. dem subglottalem System (teil davon ist die Lunge). Die Lunge und der Kehlkopf werden dabei über motorische Bewegung angeregt. Genauer gesagt, die Lunge stellt die Luft bereit, welche vom Kehlkopf benötigt wird um angeregt zu werden. Die Lunge und der Kehlkopf zusammen sorgen somit für die *Anregung*. Je nach "Form" des Vokaltrakts (Rachen-, Mund- und Nasenraum) kommt es durch die Anregung zu einer anderen *Artikulation*. Dabei werden Töne erzeugt und "ausgegeben". Aneinander gereihte Töne ergeben dadurch Sprache. Die verschiedenen Töne produzieren verschiedene Spektrale. Für bestimmte Laute bzw. Vokale lassen sich typische Muster im Spektralbereich finden und somit die Vokale unterscheiden.

4.1 Speech Recognition

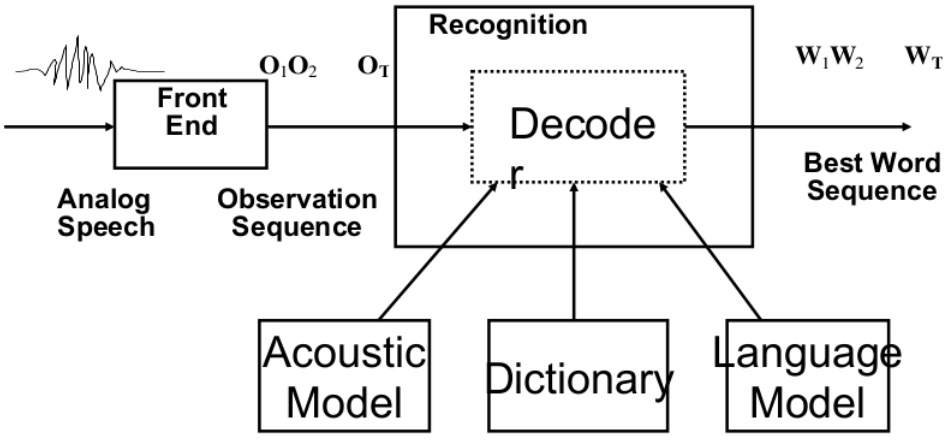


Abbildung 1: Komponenten eines Sprach Erkenners

Auf dem obigen Bild sind die einzelnen Komponenten eines Sprach Erkenners zu sehen. Links kommts das *analoge Signal* in das Front End rein und wird digitalisiert. Dazu wird es mittels Fouriertransformation in den Spektralbereich umgewandelt. Aus dem Frontend kommt die *observation sequence* heraus und wird an den *Decoder* weitergegeben. Im Decoder passiert die eigentliche Sprach Erkennung. Dazu wird ein *acoustic model*, ein *dictionary* und ein *language model* verwendet. Die einzelnen Komponenten werden wir uns später noch genauer anschauen. Der Decoder gibt am Ende die *best word sequence* aus. Das Ziel des Sprach Erkenners ist es aus einer gegebenen akkustichen Daten $A = a_1, a_2, ..., a_k$ die Wortsequenz $W = w_1, w_2, ..., w_n$ zu finden, welche die Wahrscheinlichkeit

$P(W|A)$ maximiert. Dazu brauchen wir die *Bayes Regel*:

$$P(W|A) = \frac{P(A|W) \cdot P(W)}{P(A)}$$

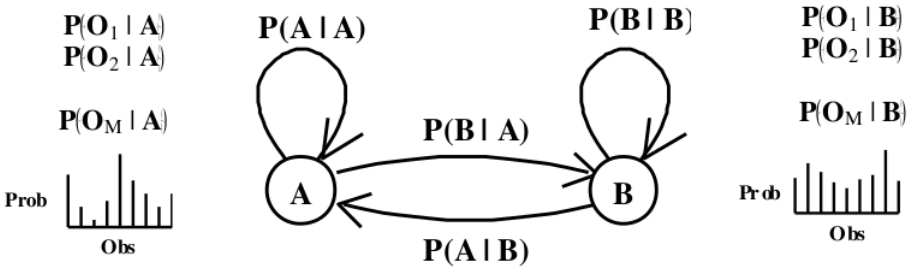
P(W|A) ist das *acoustic model*, P(W) ist das *language model* und P(A) ist konstant für einen ganzen Satz.

4.2 Acoustic Model

4.2.1 Hidden Markov Model

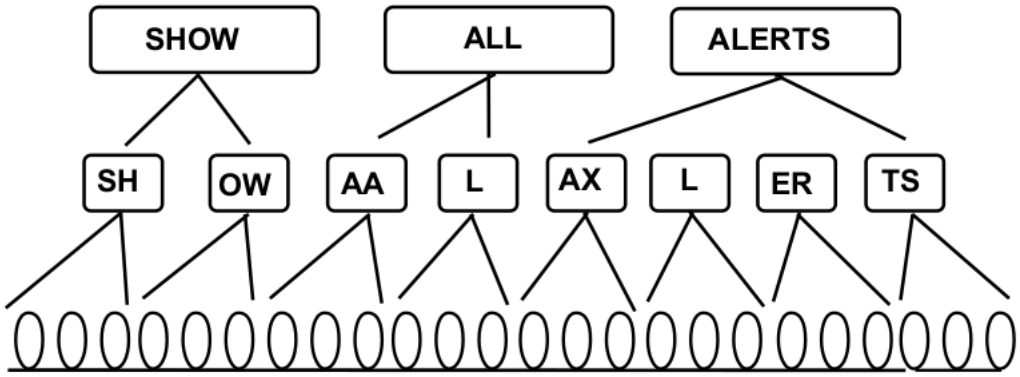
Elements:

- states: $S = \langle S_0, S_1, ..., S_N \rangle$
- transition probabilities: $P(q_t = S_i | q_{t-1} = S_j) = aji$
- emission probabilities: $P(y_t = O_k | q_t = S_j) = b_j(k)$



In der Spracherkennung spiegeln die States die phonetischen Zustände wieder (genauer gesagt besteht jedes Phonem aus drei Zuständen: beginning, middle und end). Der Decoder sucht hierbei den besten Pfad zwischen den Modellen und der Sprache. Um die Emissionswahrscheinlichkeit zu approximieren können neben HMMs alternative Methoden verwendet werden:

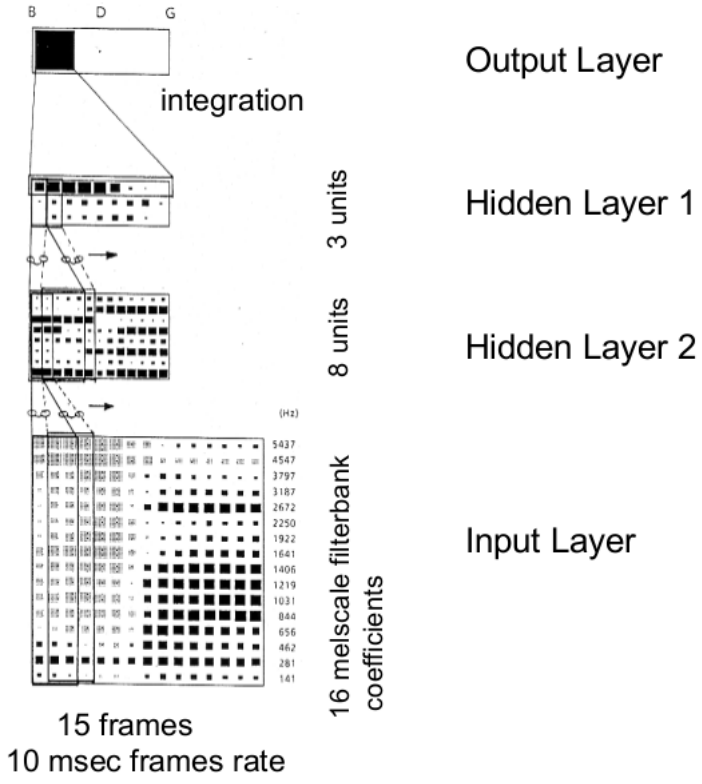
- Mixture of Gaussians Networks
- Neural Networks
- Hierarchies of Neural Networks



Die Wörter werden zerlegt in Phoneme und die Phoneme in Zustände.

4.2.2 Time Delay Neural Network (TDNN)

Ein *time delay neural netwrok* ist eine spezielle Art von neuronalem Netz, welches *time invariant* ist. Es besteht aus einem Input Layer, mehreren Hidden Layers und einem Output Layer. Als Eingabe dient eine Matrix welche *mel-scaled* ist. Die x-Achse spiegelt die Zeit wider, die y-Achse die Frequenz. Der erste Hidden Layer erhält als Eingabe ein 16 x 3 Ausschnitt der Eingabe, jeweils verschoben in der x-Richtung. Somit überlappen sich die Eingaben für den ersten Hidden Layer. Im Hidden Layer wird ebenfalls ein Fenster drüber gelegt und einem weiteren Hidden Layer als Input gegeben. Im Output Layer wird auf einzelne Buchstaben gemappt. Das bedeutet ein TDNN kann ein Muster in einem Signal finden egal wo es sich befindet. Alles was wir wissen ist, dass es darin vorkommt. Im letzten Hidden Layer werden horizontal die Aktivierungen der Neuronen angeschaut und daraus bestimmt, welches Phonem vorgekommen ist. Der letze Hidden Layer wird auch als *phoneme layer bezeichnet*. Jedoch bezieht sich das TDNN in unserem Fall auf einzelne Phoneme, jedoch werden Wörter aus mehreren Phonemen gebildet. Siehe dazu 4.3.2. TDNN sind *convolutional networks* mit einer festen Größe. Dies ist beim Übergang zwischen den Schichten zu sehen. Convolutional neuronale Netze werden in der Bildverarbeitung verwendet. Hierbei ist es so, dass z.B. Straßenschilder aufgenommen, aber nicht immer an der selben Stelle im Bild sind.



Eine Erweiterung der TDNN stellen die *frequency shifting TDNN (FSTDNN)* dar. Diese Netze sind shift invaraint in zwei Dimensionen. Das bedeute nicht nur Zeitinvariant sondern auf Frequenzinvariant. Normalerweise wird ein Netz auf eine Person trainiert und die Fehlerrate steigt, sobald eine andere Person das Netz benutzt. Die Frequenzinvarianz sorgt dafür, dass Männer, Frauen oder Kinder das selbe Nezt benutzen können und die Fehlerrate verändert sich nur gering. Dies nennt sich auch *Multi-Speaker Phoneme Recognition*. Darüber hinaus gibt es noch das Problem des Echo / der Verhallung. Wenn kein Mikrofon direkt am Mund getragen wird, wird das Signal verrauscht / verschmiert. Die Verhallung ist immer unterschiedlich und man kann kein Netz für jeden Raum bauen. Bei der Verhallung wird das ursprüngliche, gesprochene Signal mit dem Echo gefaltet. Neuronale Netze können diese Verhallung lernen und rausfiltern.

4.3 Word Model

Bisher haben wir nur die Erkennung von Phoneme behandelt. Diese müssen nun zu ganzen Wörtern zusammengebaut werden. Eine Idee wäre ein neuronales Netz für ein betimmtes Vokabular zu trainieren. Jedoch ist das Unsinn, da ein neues Wort bedeutet, dass man Beispiele sammeln muss und das Netz neu trainieren muss.

Problems:

- Time Alignment: the same word might be spoken faster / slower
- Endpoint Detection: where does one word end?
- Large Dictionaries (Training Data, Time)

4.3.1 Time Alignment

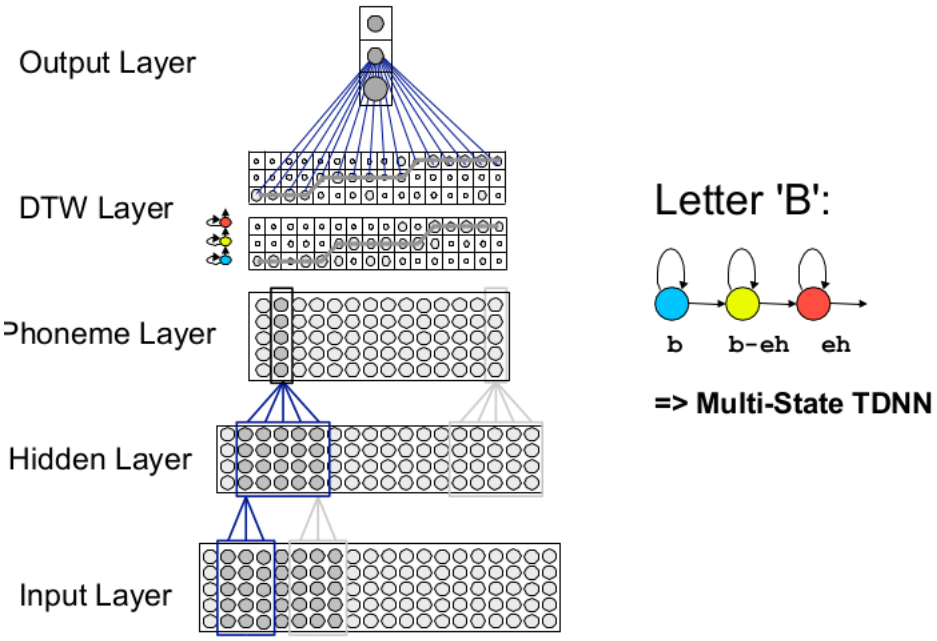
Hierbei geht es darum, dass verschiedene Sprecher unterschiedlich schnell/langsam sprechen. Irgendwie muss beim Lernen einen zeitlichen Zusammenhang herausgearbeitet werden.

Beim *Linear Sequence Alignment* wird das Referenzsignal und das unbekannte Signal linear aneinander angepasst (Stauchen / Strecken). Problem hierbei ist, dass der Mensch in der Regel nicht linear schnell redet. Wir benötigen ein **non-linear alignment**.

Eine Lösung ist das *time warping* (auch dynamic programming genannt?!?). Hierbei wird eine Verbindung zwichen zwei Sequenzen x und y gesucht. Die Achsen repräsentieren die Sequenzen, welche Vektoren sind, desen Einträge gut oder schlecht aufeinander passen. Sind die Signale aufeinander abgestimmt, kann mittels des *Viterbi-Algorithmus* die passende Sequenz Q gefunden werden, welche $P(O, Q|\lambda)$ maximiert.

4.3.2 Multi-State-TDNN

Ein MS-TDNN ist eine Erweiterung des TDNN auf Wortebene. Um ein Wort zu erkennen muss eine bestimmt Sequenz von Phonemen auftretten. Im DTW-Layer(dynamic time warping) wird die Verbindung der Phoneme vorgenommen und im Output Layer lese ich das entsprechende Wort ab. Das bedeutet aber wiederum, dass ich für jedes einzelne Wort ein Output Neuron brauche. Das ist bei großen Wörterbüchern unpraktisch. Jedoch ist das nicht so schlimm, da wir auf dem Wortlevel Trainieren können.



Ich versuche die Distanz zwischen dem richtigen Wort und dem besten, falschem Wort zu maximieren. Das heißt wir optimieren die phonetische Sequenz und nicht die Wörter an sich. Im MS-TDNN befindet sich ein Decoder. Das Alinieren der Zustände und der Feature, welche das neuronale Netz über die Zeit gelernt hat, läuft wie ein HMM mit dynamik time warping oder mit Viterbi Decoding. Der Unterschie dzu den Hypbriden, welche danach dran kommen, ist, dass das es sich hierbei um hidden Features handelt.

4.3.3 NN-HMM Hybride

Die Idee ist, dass ein neuronales Netz gebaut wird, welches die Phoneme erkennt. Wenn der Output statisch interpretiert werden kann, wird ein HMM Decoder obendrüber gebaut, der für das Alignment und die Integration der Wörter zuständig ist. Die Wahrscheinlichkeit, welche wir als Output des NN bekommen, ist nicht die richige. Um die richige Wahrscheinlichkeit zu erhalten, müssen wir die Bayes Regel umformulieren. Wir wollen P(A|W) haben aber P(W|A).
Die (log)Wortwahrscheinlichkeit besteht aus der Summe der (log)Output Activations entlang des besten Pfades (bestimmt mit DTW oder Viterbi).

Kontextabhängige Phonemmodelle (a in Kontext von P etc.)

4.3.4 Viterbi-Algorithmus

Der Viterbi Algorithmus sucht das beste Alignment zwischen allen phonetischen Sequenzen, die vorkommen können und der gesprochenen Sprache.

4.3.5 Forward-Algorithmus

4.3.6 Backward-Algorithmus

5 Learning Vector Quantization

Der initiale Ansatz ist *vector quantization*, welcher als erstes besprochen wird. Die Weiterentwicklung des vector quantization ist *learning vector quantization*, welche danach behandelt wird.

5.1 Vector Quantization

Die Idee hinter vector quantization ist, dass der *data space* mit einer kleinen Anzahl an *prototype vectors* $U = u_1, u_2, ..., u_k$ beschrieben wird. Jeder prototype vector repräsentiert eine Klasse (k Klassen). Dazu muss jeder Input Vektor $X = x_1, x_2, ..., x_m$ einem der Vektoren in U zugewiesen werden. Die prototype vectors U sind in einem *codebook* zusammengefasst. Vector Quantization ist unsupervised.

5.1.1 Applications

- multimedia compression (storage and transmission)
- dimensionality reduction
- classification

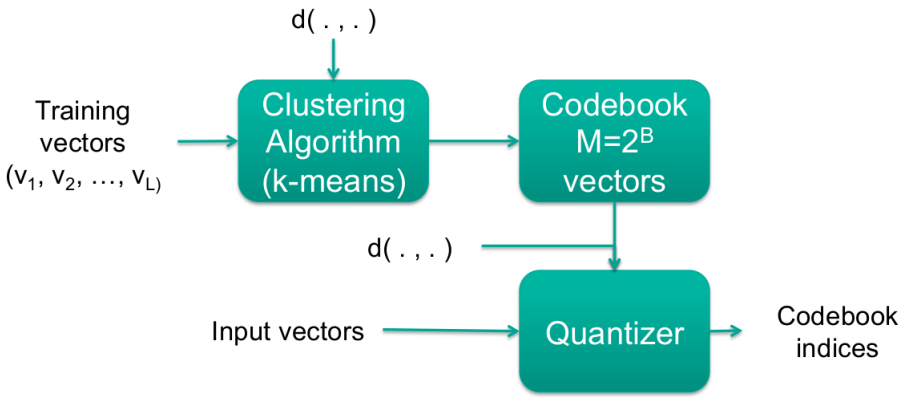
5.1.2 Training

Beim Trainieren werden die Input Vektoren mittels einem geeignetem Distanzmaß d und einem Clustering Algorithmus unterteilt. Daraus ergeben sich die Anzahl an Klassen und somit auch direkt die Anzahl an Codebuch Einträgen. Die Vektoren im Codebuch sind die *centroids* der einzelnen Regionen. Bei der eigentlichen Klassifikation (Encoding) wird für einen Input Vektor die Distanz zu den Codebuch Einträgen berechnet und bei der minimalen Entfernung der Index des prototype vectors gemerkt (bzw. übermittelt). Beim Decoding wird anhand des Index der jeweilige prototype vector ausgewählt und repräsentiert somit den Input Vektor. Der Fehler hierbei ist das Distanzmaß des Input Vektors und des prototype vectors.

5.2 Learning Vector Quantization

Da wir nun über Labels verfügen (also wissen in welche Klasse der Input Vektor gehört), lassen sich nun Wahtscheinlichkeitsverteilungen nutzen:

- $P(S_k)$: a prior Wahrscheinlichkeit der Klasse S_k
- $p(x|x \in S_k)$: bedingte Wahrscheinlichkeitsdichte
- discriminant functions: $\delta_k(x) = p(x|x \in S_k)P(S_k)$
- rate of misclassification minimized if: $\delta_c(x) = max_k\{\delta_k(x)\}$



Prinzipiell weisen wir mehrere prototype vectors den einzelnen Klasse zu. Ein Input Vector erhält die selbe Klasse wie der prototype vector, an dem der Input Vektor am nähsten ist. Da wir nun aber mehrere prototype vectors je Klasse haben, müssen wir entscheiden, welcher der *winning codebook entry* (c ist der Index) ist. Die einzelnen Algorithmen unterscheiden sich in dieser Wahl:

5.2.1 LVQ1

$c = argmin_i\{||x - m_i||\}$: Index des prototype vectors
Learning rules:

- $m_c(t+1) = m_c(t) + \alpha(t)[x(t) - m_c(t)]$: x und m_c selbe Klasse
- $m_c(t+1) = m_c(t) - \alpha(t)[x(t) - m_c(t)]$: x und m_c unterschiedliche Klasse
- $m_c(t+1) = m_i(t)$: für $i \neq c$

5.2.2 LVQ2

Die Klassifizierung ist die selbe wie bei LVQ1. Das updaten ist anders:

- m_i und m_j sind die nähsten Nachbarn von x und werden simulaten geupdated.
- x muss in ein "window" um m_i und m_j fallen.
- d_i und d_j sind die Distanzen (z.B. euklidien) zwischen x und m_i und m_j
- $min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s$ where $s = \frac{1-w}{1+w}$ (recommended window: 0.2 to 0.3)

Ergänzungen: m_i ist der Gewinner (falsche klasse) m_j zweiter gewinner, richtige klasse -> dann updaten α die Lernrate wird kleiner und kleiner

5.2.3 LVQ2.1

5.2.4 LVQ3

Ergänzen: - die Frage ist welches k das korrekte ist. Zur Not kann man die Distortion betrachten und ein anderen k wählen +- - $delta_k$ = wkeit das x zu s_k gehört - stopping rule -> fehler kleine genug oder anzahl iteration erreicht

5.2.5 OLVQ

6 Self-Organizing-Maps

6.1 Principles Of Self-Oragnized-Learning

6.1.1 Principle 1: Self-Amplification

6.1.2 Principle 2: Competition

6.1.3 Principle 3: Cooperation

6.1.4 Principle 4: Structural Information

6.1.5 Hebbian Learning

6.2 Self-Organizing-Maps

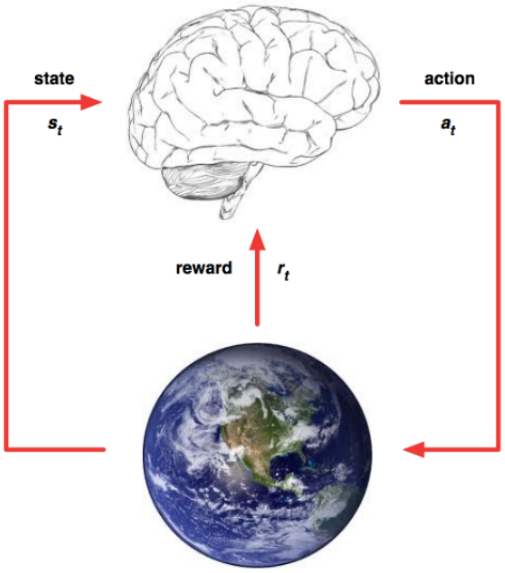
SOM theory structure learning properties VQ kernel SOM applications

7 Reinfocement Learning

- agent can act
- actions change the agent's future state
- scalar rewards for success
- (sequential decision making

Select actions to maximize the future reward.
Examples of Reinforcement Learning:

- Fly stunt maneuver in a helicopter
- Defeat the world champion at backgammon
- Manage an investment portfolio
- Control a power station
- Make a humanoid robot walk
- Play many different Atari games better than humans



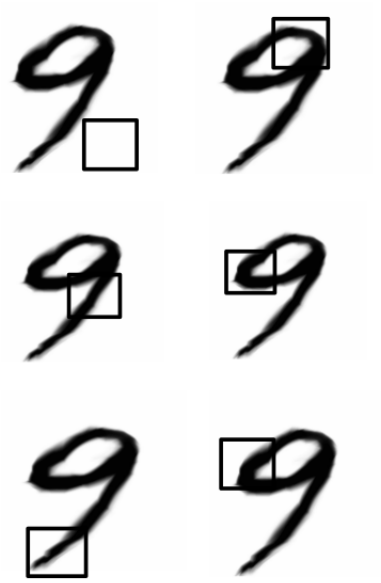
- At each step t the agent:
  - Receives state s_t
  - Receives scalar reward r_t
  - Executes action a_t
- The environment:
  - Receives action a_t
  - Emits state s_t
  - Emits scalar reward r_t
- The evolution of this process is called a Markov Decision Process (MDP)

Policy $a = \pi(s)$: probabilty distribution of actions given a state
Value function $Q^\pi(s, a)$: expected total reward from state s and action a under policy π

$$Q^\pi(s, a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ...|s, a]$$

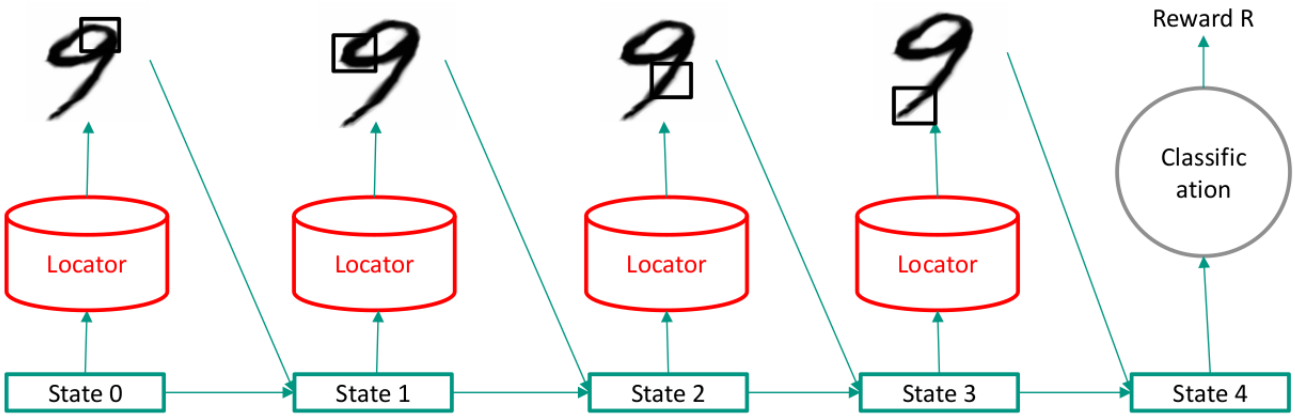
Policy-based RL: search directly for the optimal policy π (achieving maximum future reward)
Value-based RL: estimate optimal value function $Q^*(s, a)$ (maximum value achievable under any policy)
To train these networks we use *Monte Carlo methods* to learn from experience. **Policy-based Reinforcement Learning**: parameterize the policy (for example with a neural network) and based on the cumulative rewards, update the policy. We need to get the gradient of the rewards with respect to the policy:

- Policy Gradients algorithm
- Online: update after episode
- Offline: update while in episodes



- Where do you look to see if this is a 9 or an 8?
- We look at the image with some "attention" mechanism: focusing on a certain position at one time
- Changing our focus on multiple time steps until we have information to classify the image
- Reinforcement Learning can help learn where to look to classify handwritten numbers

For the image shown we can use a RNN to determine what our current state is given the last state. RL for where should we look next, given our current states.
RL gives use an reward on which we can calculate the backward pass (reward represents the gradient).



- $\frac{\Delta R}{\Delta p} = a * (R - b) * \frac{\Delta \ln f(x,p)}{\Delta p}$

  - a: Scaling factor
  - b: Baseline reward used to reduce the variance of the gradient
  - x: The sampled value (position in the image)
  - $f(x,p)$: the Probability density function, learnt by the neural network

7.1 Bellman Equation

$$Q^{\pi}(s,a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ...|s,a]$$

Recursively:

$$Q^{\pi}(s,a) = \mathbb{E}_{s'}[r_{t+1} + \gamma Q^{\pi}(s^{'}, a^{'})|s,a]$$

Optimal value function:

$$Q^{*}(s,a) = \mathbb{E}_{s'}[r_{t+1} + \gamma Q^{\pi}(s^{'}, a^{'})|s,a]$$

Value iteration solve the Ballman Equation:

$$Q_{i+1}(s,a) = \mathbb{E}_{s'}[r_{t+1} + \gamma Q_i(s^{'}, a^{'})|s,a]$$

8 Deep Learning in Computer Vision

class: was sieht man im bild? local: was sieht man und wo? (bounding boxes) detection: local + segmentation: trivial

low/mid/high - level: wie viele pixel betrachtet werden

rezeptives Feld: beim pooling wird rausgezoomt, der betrachtete Bildbereich wird vergrößert. - pooling fördert die betrachung von lokalen zusammenhängen (die ohne nvlt nicht zu erkennen sind) - überlappung fördert invarianz stationarity (translational invariance): V19F17: wenn das bild durchgeschoben wird, wird denoch die selbe wkeit detektiert (aber in unterschiedlichen Neuronen) CNN: convolutional layer, sub sampling layer, fully connected MLP multiple convolution

Image caption with attention

Calculate Parameters of NNs

9 Neural Network Applications in Machine Translation

NLP vs MT

10 Speaker Independence

Normal TDNNs are time invariance. However men, women and children differ in *frequency*. Men normally have a darker voice than women and children. Therefore we have to compensate this invariance.

Observations on TDNNs

- TDNNs develope linguistically plausible features in the hidden units
- TDNNS developed alternate internal representations that can link quite different acoustic realizations to the same higher level concept (because of multilayer arrangement)
- hidden units fire synchrony because they operate independent of precise time alignment or segmentation (time invariant)
- small network output may not be useful in complex task (but the internal abstractions may be valuable)
- complex concepts -> use stages with different knowledge
- new learning strategies should be build in existing knowledge

Model invariance

- frequency shift, tilt, compression

Variability

- adaption
  - slow adaption -> modify weights
  - fast adaption -> pretrained specific submodels
- normalization
  - environment: to the room
  - speaker: mapping new speaker to standard speaker (with standard sentences)

Combine **two standard TDNNs** to one (overall better classification): one with MSE and the other with CFM. Those combination yields the correct classification.
Even better than two TDNNs: **three TDNNS**! The third TDNN uses CE.

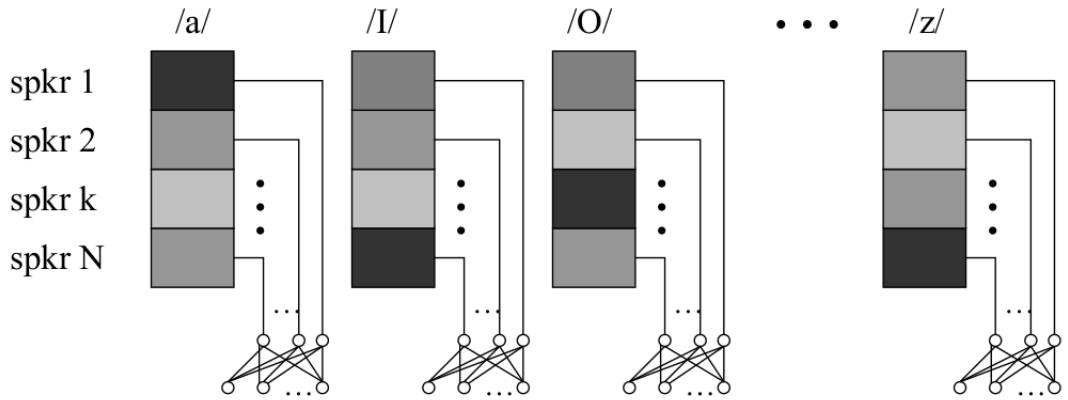
10.1 Frequency Invariance

Convolutional Acoustic Models:

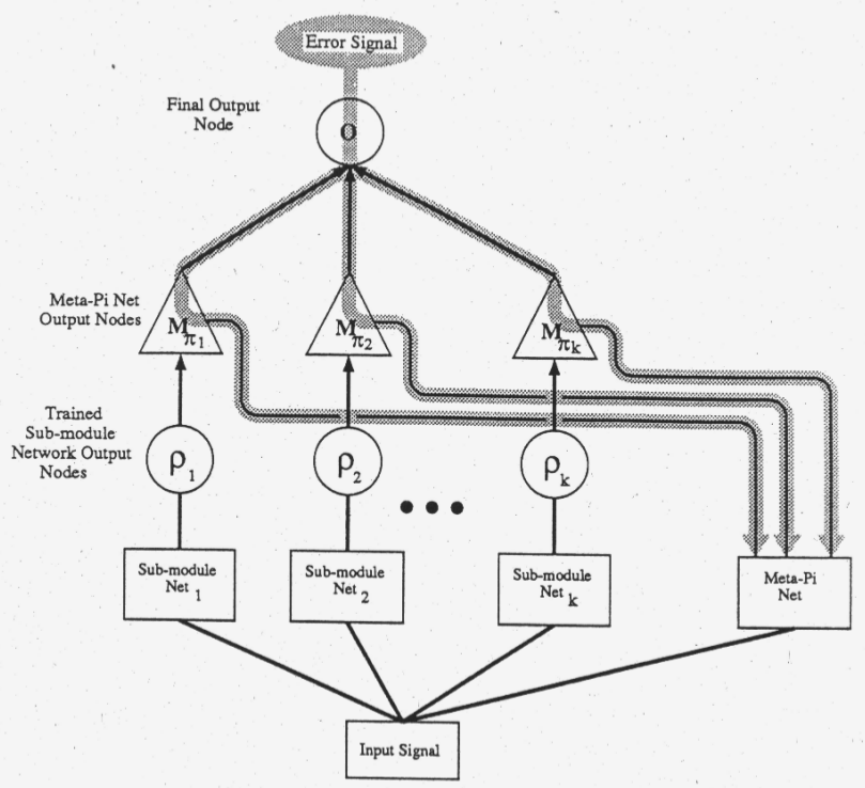
- parameter sharing across spectrum and time -> exploit 2D structure of features
- upper layer fully connected
- pooling gives more robustness and less overfitting

10.2 Multi-Speaker Reference Model

Idea: A speaker-specific reference model is composed from several well trained reference models:



Meta-Pi-Net:
A seperate neural net *Meta-Pi-Net* controls the activation / deactivation of the single nets. The whole net is a combination of one net per speaker. In general we look which speaker net is closest to the input (which nets classifies the best). The Meta-Pi-Net produces weights for each net which control how much each net is taking into account for the classification.
It is not important that the correct speaker is chosing. Sometimes a combination of two or more speaker might fit better to the input. The actual correct speaker net might not be even taking into account (results in a mixture of the other nets).



Speaker Normalization: we have speaker dependent nets. Now when we use those nets to classifier what another speaker says (no dependent net for this speaker!) the error rate is 41.9%. With 40 text-dependent training sentences the error rate is reduced to 6.8%.

i-vectors: identity vectors (i-vectors) describe the speaker-characteristic offset to an universal background model. Those i-vecotrs are used to train a recognition system.

Speaker Adaptive Training of DNN:

1. train initial DNN with (and keep it fixed)
   - SI features: fbank
   - SA features: Feature space Maximum Likelihood Linear Regression (fMLLR)
2. Train an i-vector NN
   - inputs: i-vectors
   - outputs: linear shift to the original feature vectors

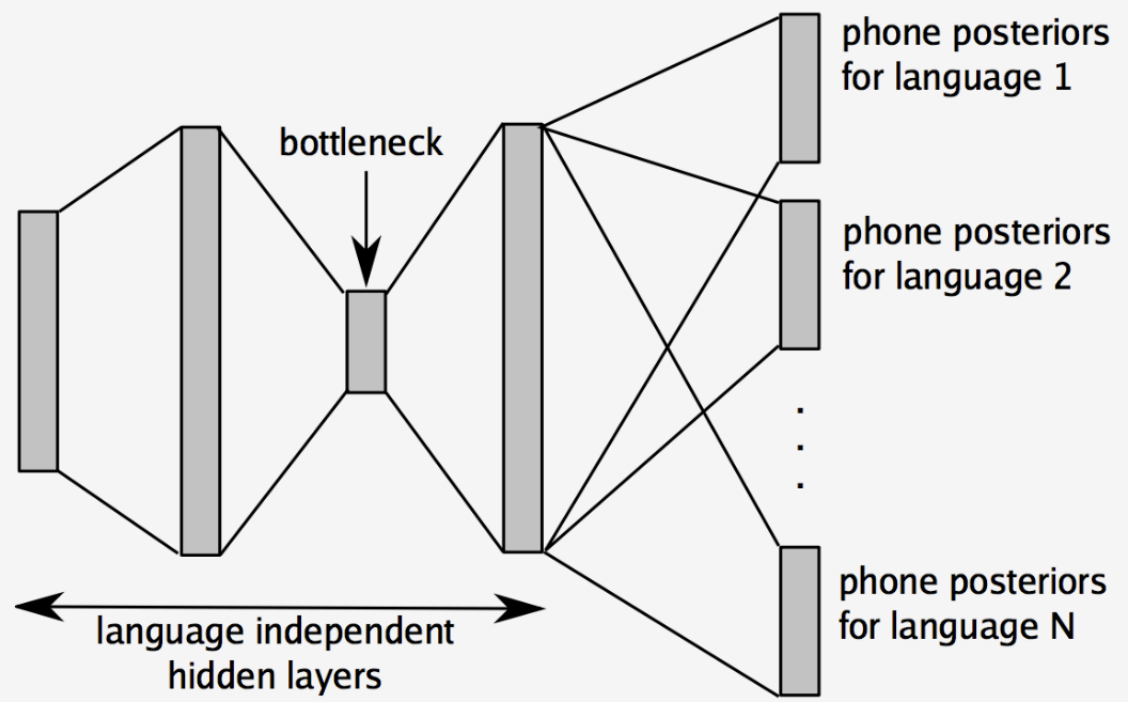
– added features become speaker-normalized

3. update the DNN in the new feature space, i-vector NN fixed -> yields SAT-DNN

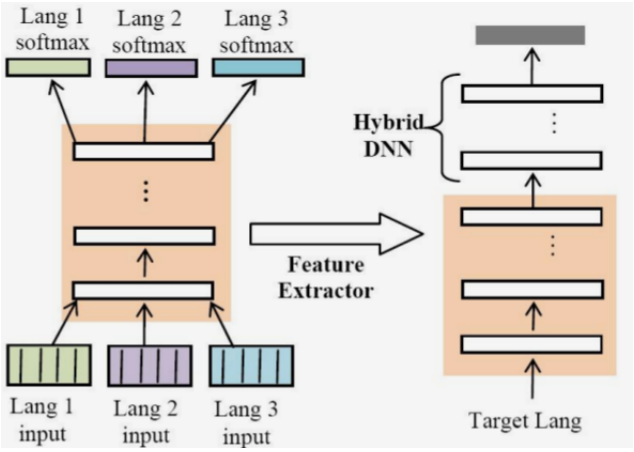
10.3 Cross-Language DNNs

Multilingual Bottleneck Features:

- Humans can only produce a finite amount of different sounds
- Subset of sounds is used in individual languages
- Some sounds are used in different languages
- Share data of the same sounds from different languages
- Extract more robust features using data with more variability



Cross-Language DNNs with Language-Universal Feature Extractors:
DNNs can be trained on mutltiple languages. The hidden layers (*language-universal feature extractor*) are used in other DNNs.



11 Hand Writing

12 Natural Language Processing

13 Gradient Optimizations and 2nd order Methods

Idea: Start from a point close to the wanted solution and iteratively move to the point that makes the gradient of the function approache zero (hence *decent*). All weights are initialised with small random numbers.
Important: update the parameters in the **opposite** direction of the gradient:

$$\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{x}, \mathbf{w})$$

13.1 Logistic Regression

$$L(\mathbf{x}, \mathbf{w}) = -\sum_k [t_k^x \log(o_k^x) + (1 - t_k^x) \log(1 - o_k^x)]$$

$$o_k^x = \sigma(\mathbf{w}\mathbf{x}_k) = \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}_k}}$$

$$\begin{aligned} \nabla_{\mathbf{w}} L &= \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{w}} \\ &= (\mathbf{o} - \mathbf{t})\mathbf{x} \end{aligned}$$

Update rule:

$$\mathbf{w} = \mathbf{w} - \eta(\mathbf{o} - \mathbf{t})\mathbf{x}$$

13.1.1 Gradient Descent - General Approach

- init $\mathbf{w}$, chose learning rate η
- iterate:
  - grad = $\mathbf{0}$
  - iterate for every instance of the training data $x \in X$:
    - compute output $o_x = \sigma(\mathbf{w}\mathbf{x})$
    - compute error $E = t_x - o_x$
    - $grad = grad + E\mathbf{x}$
  - update $\mathbf{w} = \mathbf{w} + \eta grad$
- until a convergence condition is reached (small error or all epochs done)

13.1.2 Batch Gradient Descent

Works like the normal gradient descent but the weights are updated every **epoch**!

- - iterate whole dataset to perform one update
- - slow and memory-intensive
- - can not be used for *online* training

13.1.3 Stochastic Gradient Descent

Update the weights right after we have seen *one training instance*. Before each epoch *shuffle* the training set!

- + converges much faster
- + no huge reuqiremen of memory
- + can be used for online training
- - approximation of the gradient
- - high variance in updating

13.1.4 Mini-Batch Gradient Descent

Compromise between Batch and Stochastic Gradient descent. Perform one weight update after k training instances (called 1 **iteration**).

- + faster than batch gradient descent
- + overcome memory intensity (depends on k)
- + can be used for online training
- + Faster and more stable than Stochastic Gradient Descent
- + fewer smaller weight updates
- very easy to parallelze

13.2 Learning Rate Scheduling

Decay learning rates, making it smaller as the training is going. From time step t_d (the $t_d^{th} iteration$), calculate: $\eta_{t+1} = \beta\eta_t$. Choosing learning rates and learning rate scheduling can be tricky.

13.2.1 Adagrad - ADAptive GRADient method

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{G_T + \epsilon}} \nabla_{\mathbf{w}} L(\mathbf{x}, \mathbf{w}_{t-1})$$

$G_t \in \mathbb{R}^{d \times d}$: diagonal matrix, each diagonal entry is the sum of the squares of the gradients with respect to w_i up to time step t
ϵ: small smoothing term that avoids division by zero

- + good for sparse data
- + no manual tuning of the learning rate (normally $\eta = 0.01$, $\epsilon = 1e^{-8}$)
- - have to store G_t
- - G_t is accumulated $\Rightarrow$ adaptive learning rate is smaller over time

13.2.2 Adadelta

Same as Adagrad but only stores a limited history of the gradients (normally 2).

$$\begin{aligned}
\mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{RMS[\Delta w]_{t-1}}{RMS[g]_t} g_t \\
g_t &= \nabla_{\mathbf{w}} L(\mathbf{x}, \mathbf{w}_{t-1}) \\
RMS[\Delta w]_t &= \sqrt{E[\Delta w^2]_t + \epsilon} \\
E[\Delta w^2]_t &= \gamma E[\Delta w^2]_{t-1} + (1 - \gamma) \Delta w_t^2
\end{aligned}$$

- das $E[\Delta w^2]_t$???

- + not memory intensiv
- + no picking of the learning rate needed

13.2.3 RMSprop

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

13.2.4 Adam - ADAptive Moment estimation

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\widehat{\mathbf{v}}_t + \epsilon}}\widehat{\mathbf{m}}_t$$
$$\widehat{\mathbf{m}}_t = \frac{m_t}{1-\beta_1^t}$$
$$\widehat{\mathbf{v}}_t = \frac{v_t}{1-\beta_2^t}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

-beta1, beta2??

- + faster convergence
- - pratically more overfitting
- - needs resetting
- - needs to store to matrices of past gradients

14 Error Functions

14.1 Binary Cross Entropy or Negative Log Likelihood

$$L(\mathbf{x}, \mathbf{w}) = -\sum_k [t_k^x \log(o_k^x) + (1 - t_k^x) \log(1 - o_k^x)]$$

t_k^x: target label of instance x
o_k^x: real output

$$o_k^x = P(t_k^x = 1 | x_k; w) = g(\mathbf{w} \cdot \mathbf{x}_k 1 - o_k^x \qquad = P(t_k^x = 0 | x_k; w) = 1 - g(\mathbf{w} \cdot \mathbf{x}_k$$

$g(z)$: sigmoid functions (or Logistic functions)
Mean Square Error Cross Entropy Classification Figure of Merit

15 Activation Functions

The activation functions should have the following properties:

- continous
- bounded
- monotonically increasing
- differentiable

15.1 Linear Function

$$g(x) = ax$$

15.2 Logistic Function

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

15.3 Hyperbolic Tangent function

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

max threshold ReLu