Multilingual NLP

3. Training Neural Language Models

Prof. Dr. Goran Glavaš
Center for AI and Data Science (CAIDAS), Uni Würzburg

Image: Alexander Mikhalchyk

After this lecture, you'll...

- Understand how neural LMs unify tackling of various NLP tasks
- Know the common building blocks of neural LMs
- Understand how we train deep NNs (i.e., optimize their parameters)
- Know what „dropout" is

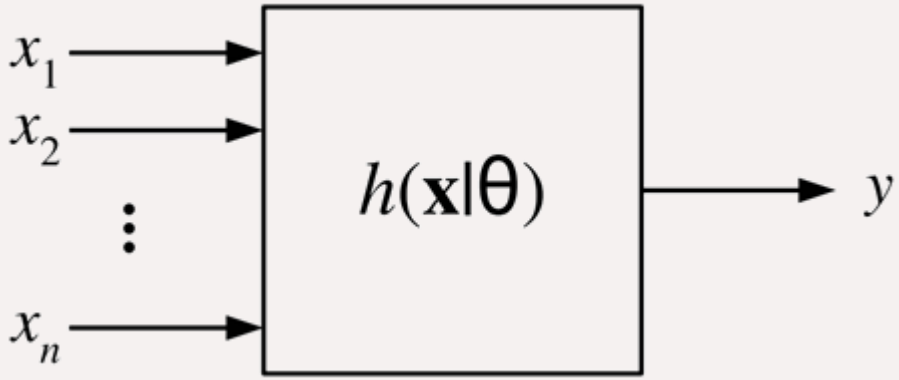
Content

- **Uniformity of NLP with Neural LMs**
- Training Neural LMs
  - Gradient Descent & Backpropagation
- Adaptive Optimization
  - Momentum, AdaGrad, RMSProp, Adam
- Dropout

Recap: (Supervised) Machine Learning

(Supervised) machine learning always has **three components**:

1. A model $h(\mathbf{x}|\boldsymbol{\theta})$: defines how the output is computed from input **x**
   - In deep learning models are highly parametrized compositions of non-linear functions (each individual function is a „layer")
     - **θ** – model's parameters



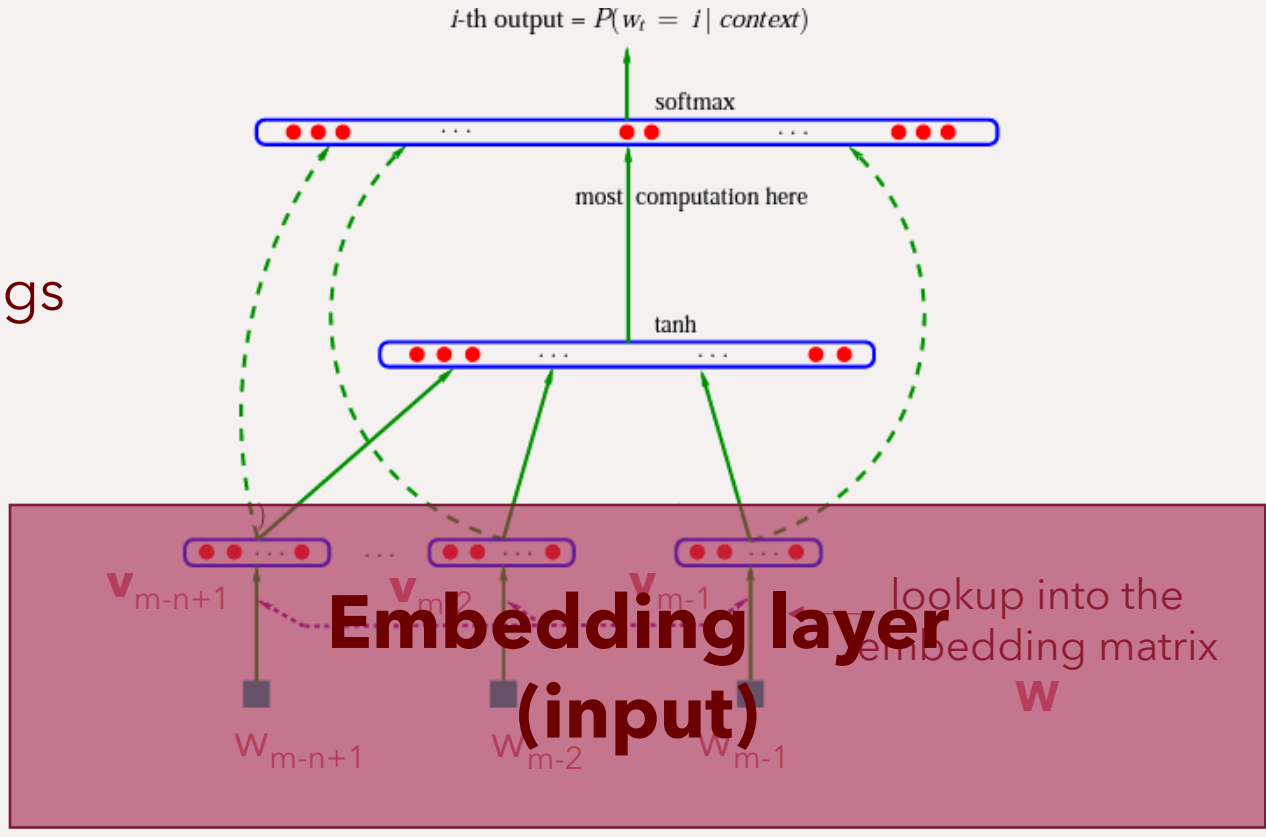
Neural Language Modeling

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, *3*, 1137-1155.

- Input: concatenation of embeddings of context words

$\mathbf{x} = \mathbf{v}_{m-n+1} \oplus \ldots \mathbf{v}_{m-2} \oplus \mathbf{v}_{m-1}$

- $\mathbf{x}$ is of length $(n-1)d$



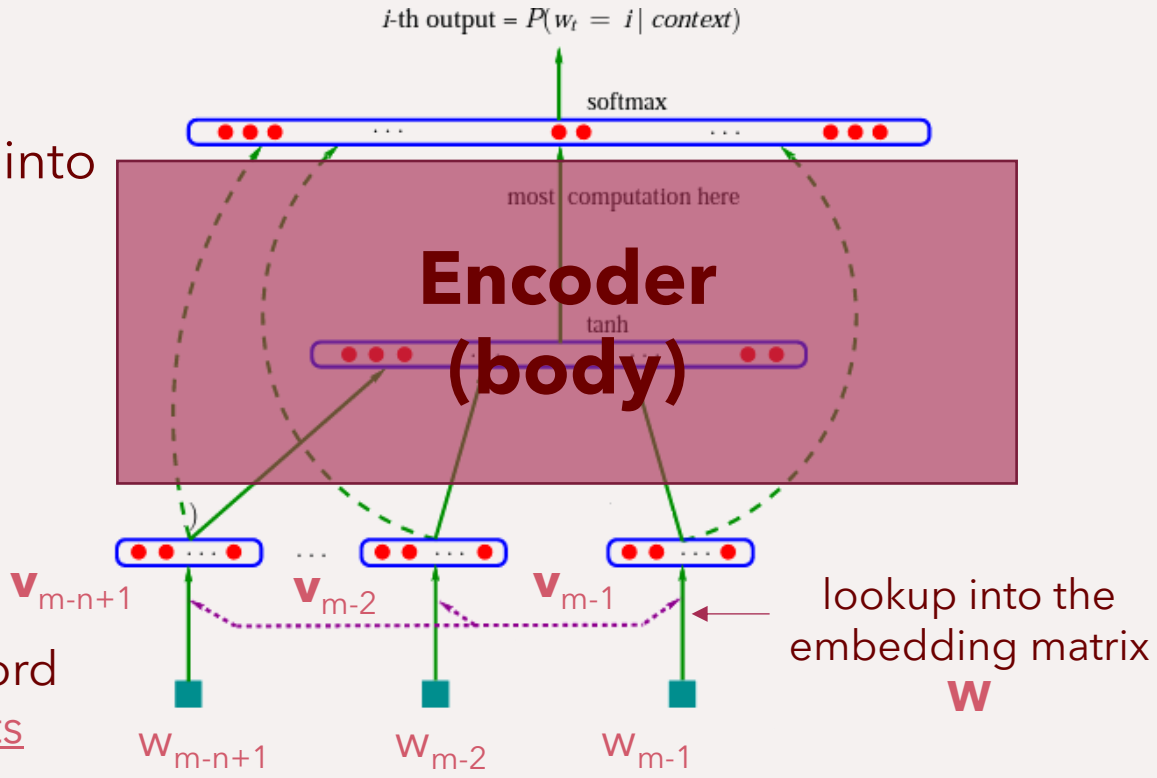
Neural Language Modeling

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, *3*, 1137-1155.

$\hat{\mathbf{y}} = \mathbf{W_2}\, tanh(\mathbf{W_1 x} + \mathbf{b_1}) + \mathbf{W_3 x} + \mathbf{b_2}$

- Layer #3: parallel linear up-projection of **x** into a vector of length |V| (vocabulary size)
  - $\mathbf{x}^{(3)} = \mathbf{W_3 x}$
  - This we will call „residual connection"
  - $\mathbf{W_3} \in \mathbb{R}^{|V| \times (n-1)d}$

- Finally, $\hat{\mathbf{y}} = \mathbf{x}^{(1)} + \mathbf{x}^{(2)} + \mathbf{x}^{(3)}$
  - Vector of |V| scores, one for each vocab. word
  - These unnormalized scores are called logits



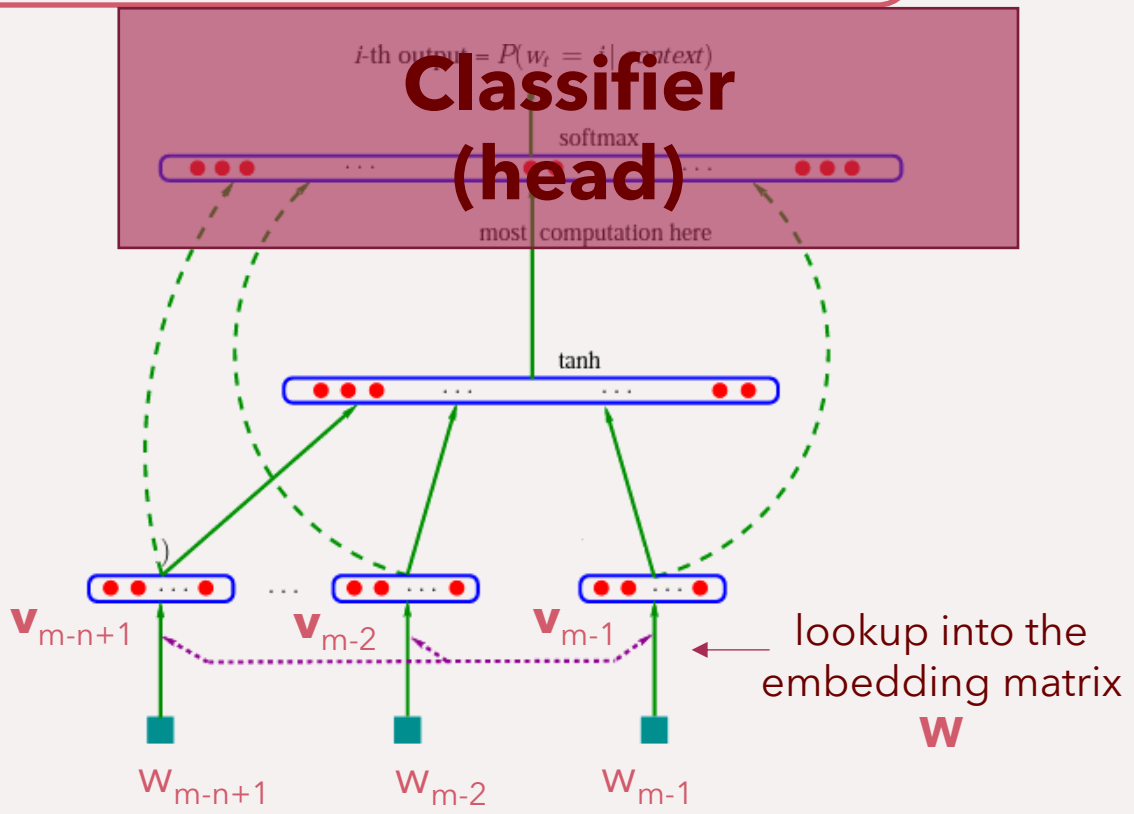
Neural Language Modeling

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, *3*, 1137-1155.

$$\hat{\mathbf{y}} = \mathbf{W_2}\, tanh(\mathbf{W_1 x} + \mathbf{b_1}) + \mathbf{W_3 x} + \mathbf{b_2}$$

- $\hat{\mathbf{y}} \in \mathbb{R}^{|V|}$ is a vector of *logits*
- But we need $P(w \mid w_{m-n+1} \ldots w_{m-1})$ for each word w from the vocabulary V
- Need to convert $\hat{\mathbf{y}}$ into a probability distribution
- **Softmax** function:

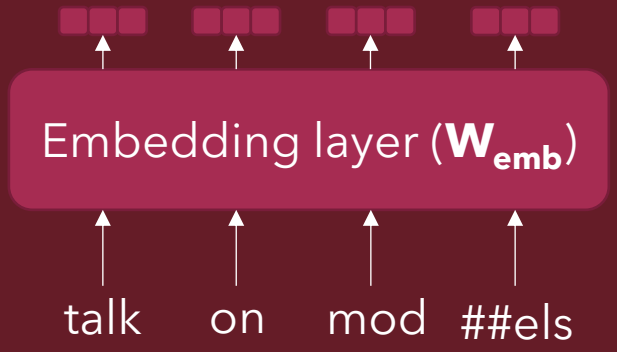
$$\hat{y}_i \rightarrow \frac{e^{\hat{y}_i}}{\sum_{j=1}^{|V|} e^{\hat{y}_j}}$$



Neural LMs

- All neural LMs have the same three main components

1. **Embedding layer** („feet" of the model)
   - Embedding matrix $\mathbf{W_{emb}}$ contains embeddings for all terms from vocabulary **V**
   - Input text is tokenized into tokens t_1, ..., t_T
   - Embedding layer is simply a lookup into $\mathbf{W_{emb}}$, fetches embeddings $\mathbf{t_1}$, $\mathbf{t_2}$, ..., $\mathbf{t_T}$

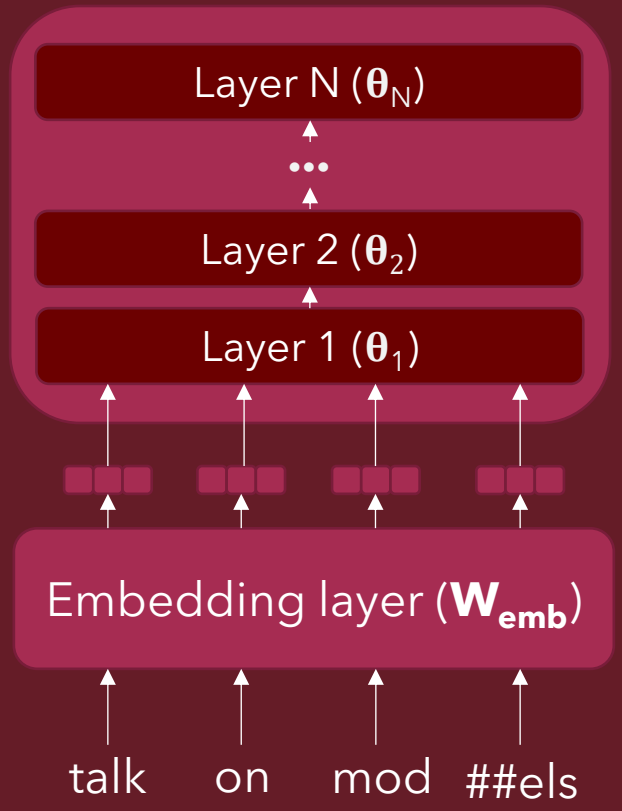


Neural LMs

- All neural LMs have the same three main components

2. Encoder (body of the model)

- Conceptually: just a parametrized function
- Reality: very complex and highly parametrized function
- **Composition** of smaller (typically non-linear) parametrized functions, called **layers**



Neural LMs

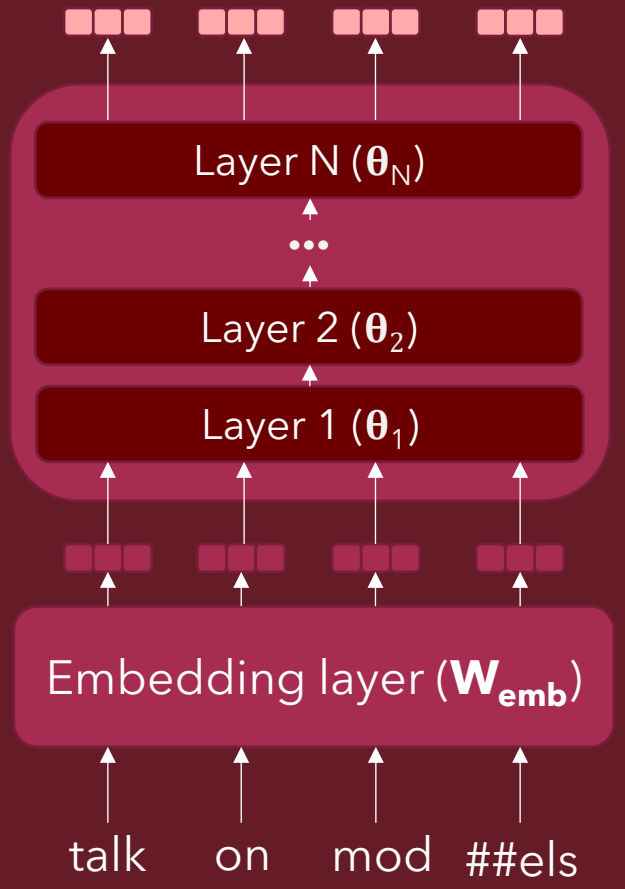
- All neural LMs have the same three main components

2. Encoder (body of the model)

- $\mathbf{t}_1^N, \mathbf{t}_2^N, \ldots, \mathbf{t}_T^N = \mathbf{enc}(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_T \mid \boldsymbol{\theta}_{enc})$
- Encoder: a composition of layer functions

$$\mathbf{enc}(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_T \mid \boldsymbol{\theta}_{enc})$$
$$= lay_N(\mathbf{t}_1^{N-1}, \mathbf{t}_2^{N-1}, \ldots, \mathbf{t}_T^{N-1} \mid \boldsymbol{\theta}_N)$$
$$= lay_N(lay_{N-1}(\mathbf{t}_1^{N-2}, \mathbf{t}_2^{N-2}, \ldots, \mathbf{t}_T^{N-2} \mid \boldsymbol{\theta}_{N-1}) \mid \boldsymbol{\theta}_N)$$
$$= \ldots$$
$$= lay_N(lay_{N-1}(\ldots(lay_1(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_T \mid \boldsymbol{\theta}_1)\ldots \mid \boldsymbol{\theta}_{N-1}) \mid \boldsymbol{\theta}_N)$$

Encoder parameters: $\boldsymbol{\theta}_{enc} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_{N-1}, \boldsymbol{\theta}_N\}$

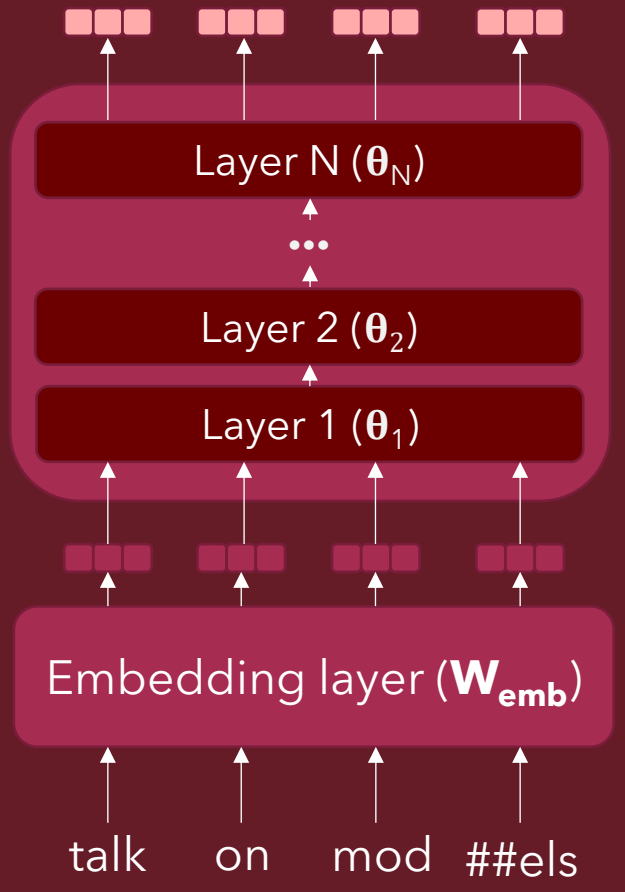


Neural LMs

- All neural LMs have the same three main components

2. Encoder (body of the model)

- $\mathbf{t}_1^N, \mathbf{t}_2^N, \ldots, \mathbf{t}_T^N = lay_N(lay_{N-1}(\ldots(lay_1(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_T \mid \boldsymbol{\theta}_1)\ldots \mid \boldsymbol{\theta}_{N-1}) \mid \boldsymbol{\theta}_N)$
- In most modern neural LMs, layers are identical
  - Same parametrized function, $lay_N = lay_{N-1} = \ldots = lay_1 = lay$
- But each layer has its own set of parameters!
  - Parameters, in principle, not shared across layers
  - Encoder parameters: $\boldsymbol{\theta}_{enc} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_{N-1}, \boldsymbol{\theta}_N\}$
- Each layer itself is again a composition of parametrized functions, which we'd commonly call sublayers

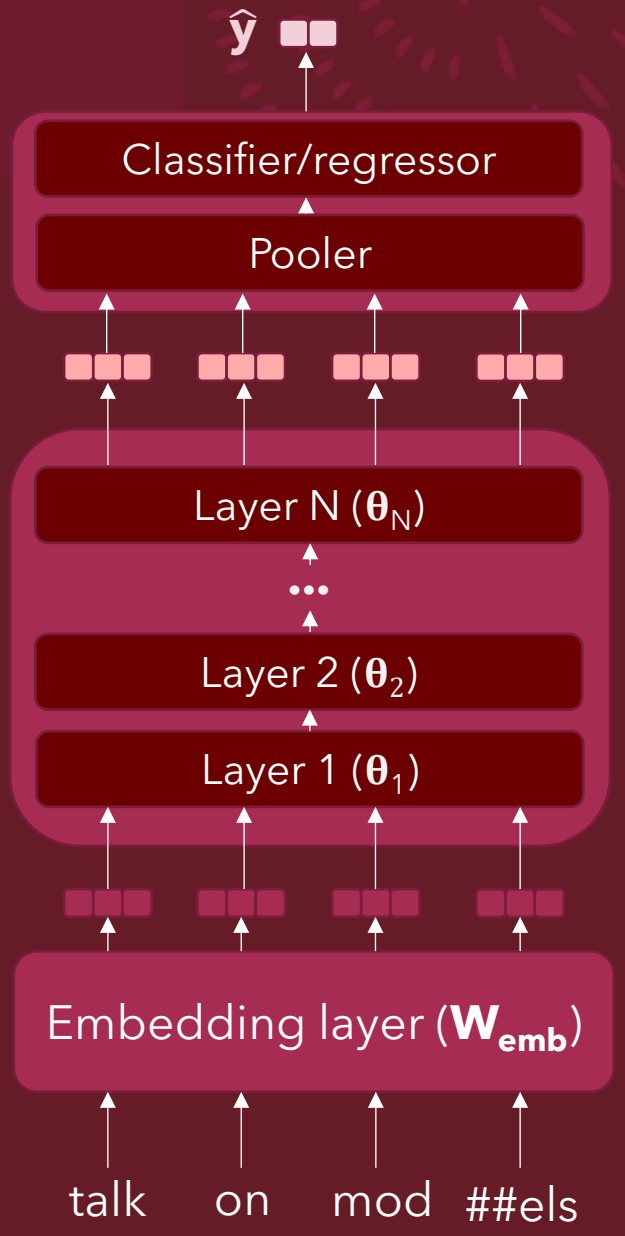


Neural LMs

- All neural LMs have the same three main components

3. **Classifier** (or regressor; head of the model)
   - Its architecture depends on the concrete task for which we're training the neural LM model
   - Typically has two sub-components
   1. **Pooling layer** (or pooler) produces an aggregate representation of the input
      - Commonly a parameterless function (e.g., average)

        $\mathbf{x} = \text{agg}(\mathbf{t}_1^N, \mathbf{t}_2^N, \ldots, \mathbf{t}_T^N)$
      - In token-level tasks, there's typically no pooling



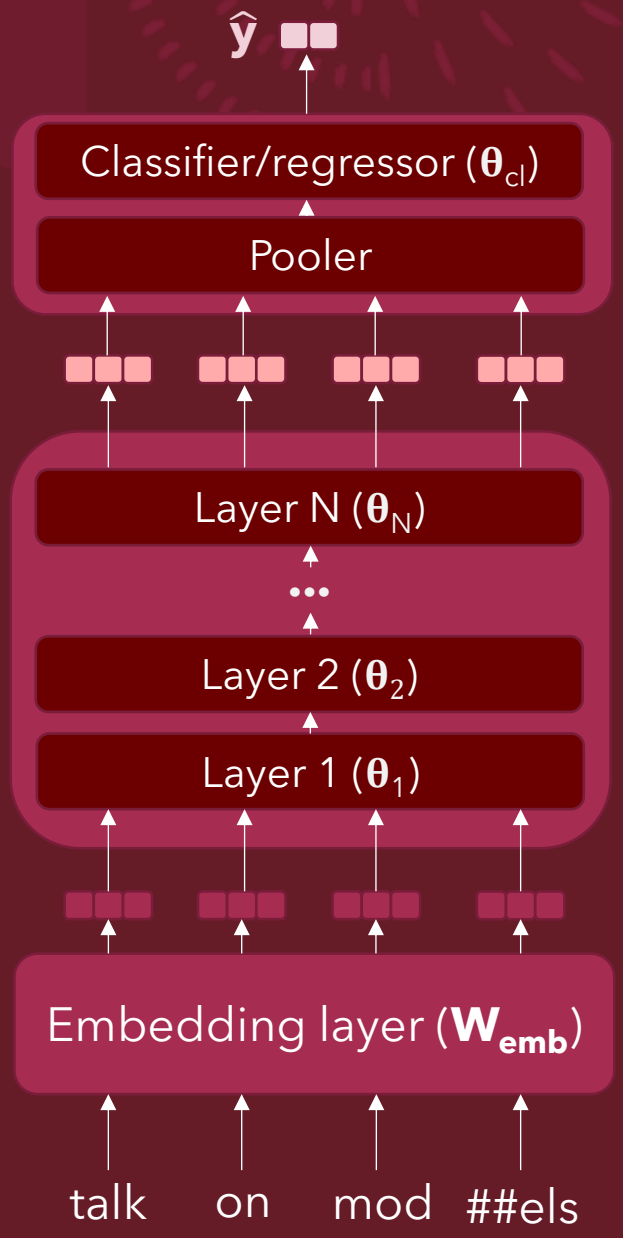
Neural LMs

- All neural LMs have the same three main components

3. **Classifier** (or regressor; head of the model)
   - Its architecture depends on the concrete task for which we're training the neural LM model
   - Typically has two sub-components
   2. **Classification/regression model**
      - We usually don't need many parameters in the classifier.
      - Q: Why?
      - A single-hidden-layer feed-forward neural network

$$\hat{\mathbf{y}} = \textit{classifier}(\mathbf{x} \mid \boldsymbol{\theta}_{cl}) = \mathbf{W_2}\ \textit{tanh}(\mathbf{W_1 x} + \mathbf{b_1})$$

      - Classifier's parameters: $\boldsymbol{\theta}_{cl} = \{ \mathbf{W_1}, \mathbf{b_1}, \mathbf{W_2} \}$
      - C = number of classes (in regression tasks, C=1), $\mathbf{W_2} \in \mathbb{R}^{h \times C}$



Recap: (Supervised) Machine Learning

(Supervised) machine learning always has **three components**:

1. (Neural LM) Model
   - Embedding layer (feet) + Encoder (body) + Classifier (head)
   - All model's parameters:

$$\boldsymbol{\theta} = \{\mathbf{W}_{\text{emb}}, \boldsymbol{\theta}_{\text{enc}}, \boldsymbol{\theta}_{\text{cl}}\}$$

2. An objective function
   - Depends on the nature of the classification/regression task

3. Optimization algorithm
   - End-to-end training/optimization: we optimize all parameters $\boldsymbol{\theta}$ during one (the same) training/optimization procedure

Uniforming NLP with Neural LMs

- (One of the) problem(s) of traditional NLP
  - Different model for each task
  - Task-specific features precomputed from the symbolic text input

- Neural LMs make NLP much more **uniform**
  - Every NLP task benefits from semantic representations of input (embedding layer)
  - Every NLP task benefits from contextualization of token embeddings against each other (encoder)
  - Embedding layer & encoder: the same, regardless what the task is
  - Classifier: depends on the task-type (but not concrete task itself)

Uniform NLP with Neural LMs

- The vast majority of NLP tasks fall into one of three categories
  - Sequence classification
  - Token classsification
  - Text generation

- Notable exceptions (need task-specific heads)
  - Syntactic parsing
  - Coreference resolution

Sequence classification

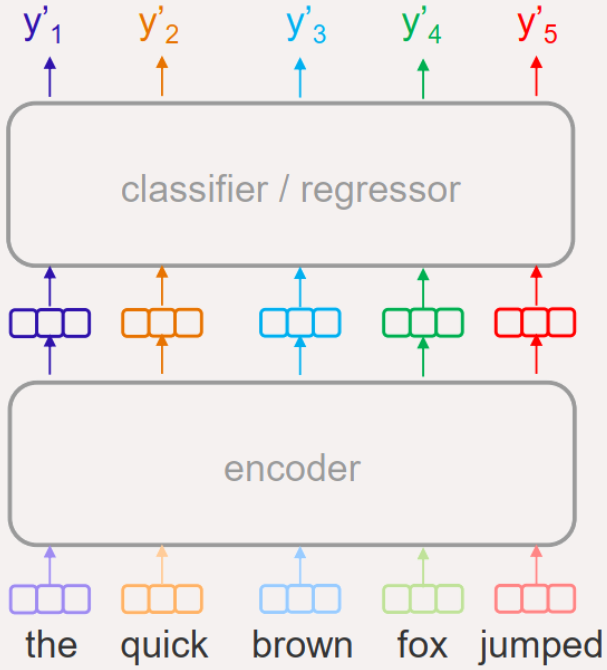
- **Sequence classification** (or regression) denotes tasks in which a label (class or score) is to be assigned to the whole input text
- Examples:
  - Classifying product reviews for sentiment
  - Topical classification of news stories
  - Predicting semantic similarity for a pair of sentences/texts
  - Natural language inference: predict if one sentence is logically entailed by the other sentence
- We pool the encoded token representations and feed the aggregation into the classifier/regressor
  - Averaging is the most commonly used pooling function

$$\mathbf{x} = \text{agg}(\mathbf{t}_1^N, \mathbf{t}_2^N, \ldots, \mathbf{t}_T^N) = \frac{1}{T}\sum_{i=1}^{T} \mathbf{t}_i^N \qquad \hat{\mathbf{y}} = \textit{classifier}(\mathbf{x}|\boldsymbol{\theta}_{\mathbf{cl}})$$

Token classification

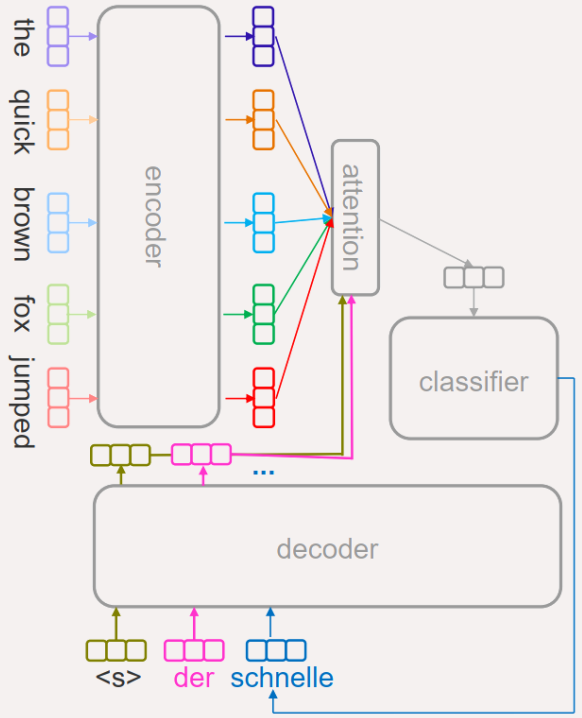
- **Token-level classification** (or regression), also known as sequence labeling, denotes tasks in which a label (class or score) is to be assigned to each input token
- Examples:
  - Part-of-speech tagging
  - Named entity recognition
  - Any of the other IE tasks where we need to extract the span of tokens that represent a concept instance
- No pooling, the encoded representation of each token is directly fed to the classifier

$$\hat{\mathbf{y}}_i = classifier(\mathbf{t}_i^N | \boldsymbol{\theta}_{cl}), i \in \{1, ..., T\}$$



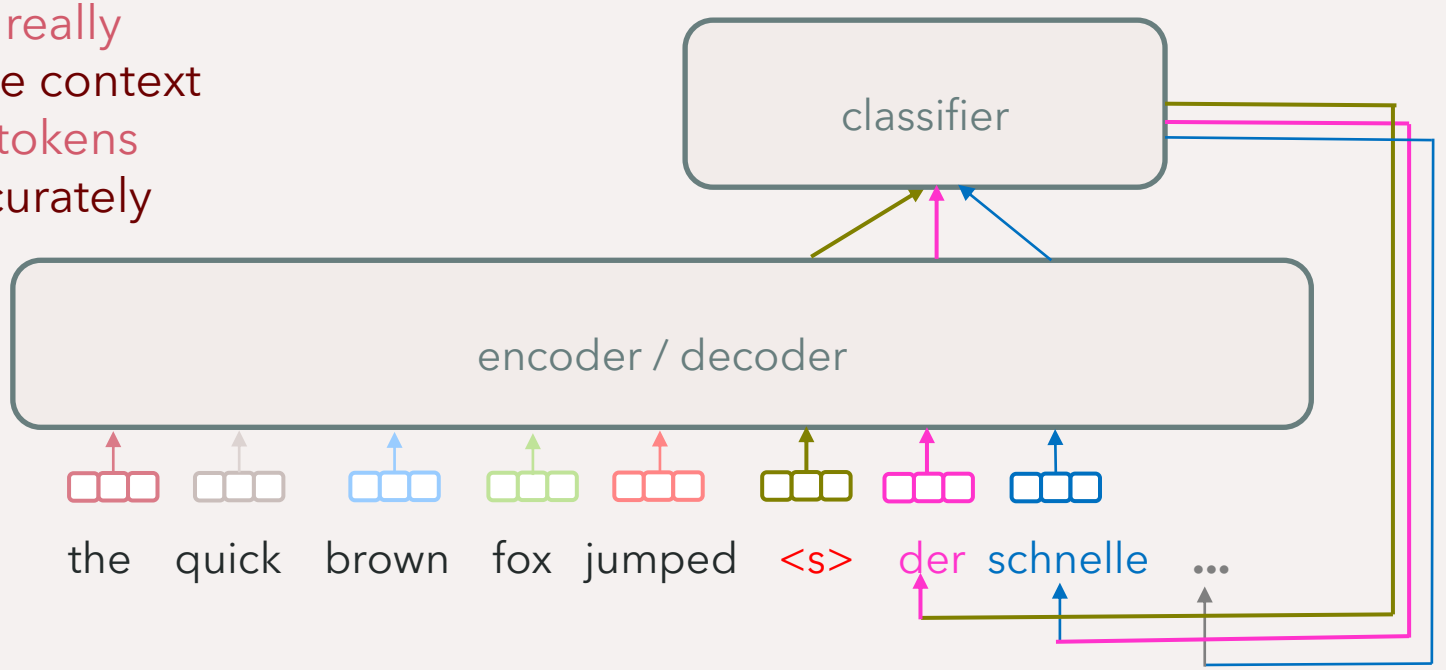
Generation tasks

- **Text generation** denotes tasks in which the model (neural LM) is to generate text starting from some given/preceding context
- Example tasks:
  - Text summarization
  - Machine translation
  - Data-to-text generation
  - Dialogue („Conversational AI")
- Traditional neural generation:
  - What we called „encoder" in generic neural LM, now becomes a „decoder"
  - Pooling across the representations of the context and previously generated tokens



Generation tasks

- **Text generation** denotes tasks in which the model (neural LM) is to generate text starting from some given/preceding context
- Example tasks:
  - Text summarization, Machine translation, Data-to-text generation, Dialogue
- Modern neural generation:
  - Powerful neural LMs: we don't really need a separate encoder of the context
  - Context just fed as preceding tokens
    - LLMs can semantically accurately
    - encode long contexts
    - (GPT-4o: 128K tokens)
  - In generation tasks commonly C = |V|
    - „classes" are tokens from the vocabulary



Content

- Uniformity of NLP with Neural LMs
- **Training Neural LMs**
  - **Gradient Descent & Backpropagation**
- Adaptive Optimization
  - Momentum, AdaGrad, RMSProp, Adam
- Dropout

Training Objectives

- **Objective functions** with neural LMs
  - Loss functions that we're trying to minimize
- Classification
  - Binary cross-entropy (for one-class binary classification)
  - Negative log-likelihood (or cross-entropy loss)
- Regression
  - Mean Squared Error

Training Objectives

- **x** denotes the representation being classified
  - Sequence or token encoding, output of the encoder
  - Of hidden size dimension h
- Binary cross-entropy loss (for one-class binary classification)
  - An instance being classified either belongs to the class of interest (is c) or it doesn't (not c)
  - We only care about c, „not c" is not a „real" class
    - E.g., spam detection – we care about recognizing spam
  - The classifier is essentially logistic regression
    - **Prediction**: $\hat{y} = \sigma(\mathbf{w}^T\mathbf{x} + b); \mathbf{w} \in \mathbb{R}^h \quad b \in \mathbb{R}$
      $= 1/(1 + e^{-(\mathbf{w}^T\mathbf{x} + b)})$
    - **Loss**: $L_{BCE} = -y \ln \hat{y} - (1- y) \ln (1-\hat{y})$

Training Objectives

- **x** denotes the representation being classified
  - Sequence or token encoding, output of the encoder
- Negative log-likelihood (for multi-class classification)
  - Aka (regular) cross entropy loss
  - The classifier is essentially softmax regression
    - **Prediction**: $\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$; $\mathbf{W} \in \mathbb{R}^{C \times h}$, $\mathbf{b} \in \mathbb{R}^{C}$
    - **Loss**: $L_{NLL} = -\sum_{i=1}^{C} y_i \ln \hat{y}_i$
      - $y_{i=c} = 1$ only for the index i that corresponds to the actual class c of the example, all other $y_{i \neq c} = 0$
      - So, $L_{NLL} = -\ln \hat{y}_{i=c}$

Training Objectives

- **x** denotes the representation being classified
  - Sequence or token encoding, output of the encoder
- (Mean) Squared error (for regression)
  - The „regressor" outputs a score
    - **Prediction**: $\hat{y} = g(\mathbf{w}^T\mathbf{x} + b); \mathbf{w} \in \mathbb{R}^h, b \in \mathbb{R}^C$
      - g is the score normalization function, identity function if no normalization
    - **Loss**: $L_{MSE} = (y - \hat{y})^2$

Training Objectives

- Loss functions L defined for a single training example $(\mathbf{x}, \mathbf{y})$
- But we normally do not train our neural LMs with individual examples

- **Training dataset**: $D = \{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^{N}$
  - The actual loss that we minimize is an average over losses of individual examples:
    - $L_D = \sum_{k=1}^{N} L(\hat{\mathbf{y}}_k, \mathbf{y}_k))$
      - $\hat{\mathbf{y}}_k = model(\mathbf{x}_k|\boldsymbol{\theta})$

- Model training means solving the following
  - $\hat{\boldsymbol{\theta}} = \text{argmin}_{\boldsymbol{\theta}} L_D$, which means solving $\nabla_{\boldsymbol{\theta}} L_D = 0$
  - With $model(\mathbf{x}|\boldsymbol{\theta})$ being a complex neural LM and D being a (very) large dataset, this equation clearly has no closed-form solution

Optimization algorithm

- We resort to (typically unconstrainted) numerical optimization

Numerical Optimization

Numerical optimization refers to optimizing real-valued functions $f(\boldsymbol{\theta})$: $\mathbb{R}^n \rightarrow \mathbb{R}$, $\boldsymbol{\theta} = \theta_1, \theta_2, \ldots, \theta_n \in \mathbb{R}$. This means finding values $\theta_1, \theta_2, \ldots, \theta_n$ for which f obtains the minimal or maximal value.

- Concretely, optimization of deep NNs relies on gradient-based optimization, i.e., variants of gradient descent

- **Gradient descent** – optimization algorithm that uses function differentiation (w.r.t. parameters) to find the minimum of a function

Optimization algorithm

- Our loss function L_D needs to be differentiable w.r.t. all parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, \ldots, \theta_n\}$

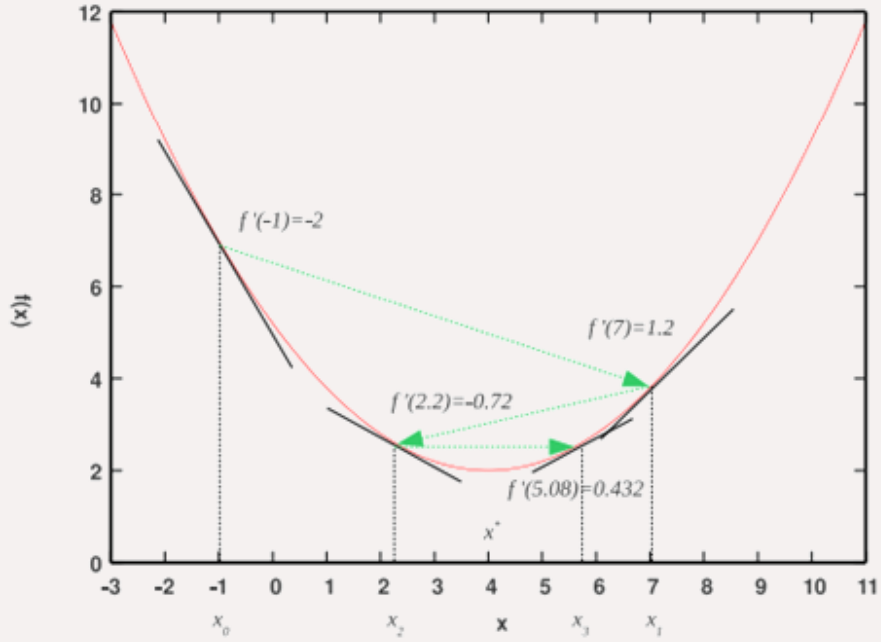
Gradient of a differentiable function

A function of multiple parameters $f(\boldsymbol{\theta} = \theta_1, \theta_2, \ldots, \theta_n)$ is differentiable if its **gradient** $\nabla_{\boldsymbol{\theta}} f$ – a vector of **partial derivatives** $\nabla_{\boldsymbol{\theta}} f = [\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \ldots, \frac{\partial f}{\partial \theta_n}]$ – exists for every point on the input domain that is $\subseteq \mathbb{R}^n$.

- If function is differentiable, then it is also continuous. Most continuous functions used in NNs are differentiable.

Gradient Descent

- **Gradient descent** is a method that moves the parameter values in the direction opposite of the function's gradient in the current point
  - This is guaranteed to lead to a **minimum** only for **convex** functions*
- Loss functions for tasks solved with neural LMs are most certainly not globally convex



Gradient Descent

Gradient Descent

Gradient descent (sometimes also called steepest descent) is an iterative algorithm for (continuous) optimization that finds a minimum of a convex (single) differentiable function.

- In each iteration GD moves the values of parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, \ldots, \theta_n\}$ in the direction **opposite** to the gradient in the current point

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}^{(k)})$$

- $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$ – value of the gradient (a vector of same dimensionality as $\boldsymbol{\theta}$) of the function f in the point $\boldsymbol{\theta}$
- η – learning rate, defines by how much to move the parameters in the direction opposite of the gradient

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (**1986**). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

- To update some parameter θ_i we need to compute in closed-form the partial derivative of the loss L_D w.r.t. θ_i : $\frac{\partial L_D}{\partial \theta_i}$
- Our L_D is a complex composition of non-linear parametrized functions
  - Because it's computed on the output of the *model*
    - $L_D = \sum_{k=1}^{N} L(\hat{\mathbf{y}}_k , \mathbf{y}_k))$
      $= \sum_{k=1}^{N} L(model(\mathbf{x}_k|\boldsymbol{\theta}), \mathbf{y}_k))$
      $= \sum_{k=1}^{N} L(lay_N(lay_{N-1}(...(lay_1(\mathbf{x}|\ \boldsymbol{\theta}_1)...|\ \boldsymbol{\theta}_{N-1})\ |\ \boldsymbol{\theta}_N) , \mathbf{y}_k))$

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (**1986**). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

$$
\begin{aligned}
L_D &= \sum_{k=1}^{N} L(\hat{\mathbf{y}}_k, \mathbf{y}_k)) \\
&= \sum_{k=1}^{N} L(model(\mathbf{x}_k|\boldsymbol{\theta}), \mathbf{y}_k)) \\
&= \sum_{k=1}^{N} L(lay_N(lay_{N-1}(\ldots(lay_1(\mathbf{x}|\ \boldsymbol{\theta}_1)\ldots|\ \boldsymbol{\theta}_{N-1})\ |\ \boldsymbol{\theta}_N)\ ,\ \mathbf{y}_k))
\end{aligned}
$$

- Let $\boldsymbol{\theta}_{ij}$ denote the j-th parameter of the i-th layer of the model
- Computing $\frac{\partial L_D}{\partial \theta_{i,j}}$ in closed form for params $\boldsymbol{\theta}_{N,j}$ of the last layer is easy
- But it gets progressively more cumbersome and difficult the „earlier" (i.e., „deeper") the layer of the parameter is

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (**1986**). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

- Computing $\frac{\partial L_D}{\partial \theta}$ in closed form gets progressively more cumbersome and difficult the earlier the layer of the parameter is

- **Backpropagation** leverages the chain rule of differentiation to avoid computation of closed-form gradients for „deeper" parameters
  - Gradients of parameters from layer K are estimated from gradients of parameters from layer K+1

$$\frac{\partial L_D}{\partial \theta_{ij}} = \frac{\partial L_D}{\partial L} \frac{\partial L}{\partial model} \frac{\partial model}{\partial lay_N} \frac{\partial lay_N}{\partial lay_{N\_1}} \cdots \frac{\partial lay_{i+1}}{\partial lay_i} \frac{\partial lay_i}{\partial \theta_{ij}}$$

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (**1986**). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

$$\frac{\partial L_D}{\partial \theta_{ij}} = \frac{\partial L_D}{\partial L}\frac{\partial L}{\partial model}\frac{\partial model}{\partial lay_N}\frac{\partial lay_N}{\partial lay_{N\_1}} \cdots \frac{\partial lay_{i\_1}}{\partial lay_i}\frac{\partial lay_i}{\partial \theta_{ij}}$$

- For the last layer:
  - $\frac{\partial L_D}{\partial \theta_{N'j}} = \underbrace{\frac{\partial L_D}{\partial L}\frac{\partial L}{\partial model}\frac{\partial model}{\partial lay_N}}_{\delta_N}\frac{\partial lay_N}{\partial \theta_{i'j}}$

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (**1986**). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

$$\frac{\partial L_D}{\partial \theta_{ij}} = \frac{\partial L_D}{\partial L}\frac{\partial L}{\partial model}\frac{\partial model}{\partial lay_N}\frac{\partial lay_N}{\partial lay_{N\_1}} \cdots \frac{\partial lay_{i\_1}}{\partial lay_i}\frac{\partial lay_i}{\partial \theta_{ij}}$$

- For layer N-1 (and then so on backwards for all layers):
  - $\frac{\partial L_D}{\partial \theta_{N\_1'j}} = \underbrace{\frac{\partial L_D}{\partial L}\frac{\partial L}{\partial model}\frac{\partial model}{\partial lay_N}}_{\delta_N}\frac{\partial lay_N}{\partial lay_{N\_1}}\frac{\partial lay_{N\_1}}{\partial \theta_{N\_1'j}}$

$\delta_{N\text{-}1}$

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (**1986**). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

$$\frac{\partial L_D}{\partial \theta_{N\_1'j}} = \delta_N \frac{\partial lay_N}{\partial lay_{N\_1}} \frac{\partial lay_N}{\partial \theta_{N\_1'j}}$$

…

$$\frac{\partial L_D}{\partial \theta_{ij}} = \delta_{i+1} \frac{\partial lay_{i+1}}{\partial lay_i} \frac{\partial lay_i}{\partial \theta_{ij}}$$

…

$$\frac{\partial L_D}{\partial \theta_{1j}} = \delta_2 \frac{\partial lay_2}{\partial lay_1} \frac{\partial lay_1}{\partial \theta_{1j}}$$

- With backprop we avoid having to explicitly compute gradient functions for all layers/parameters
- But we have to compute gradients in the inverse order of layers ☺
- Gradient of a subsequent layer needed for the computation of the gradient of the preceding layer

Stochastic gradient descent

- We never compute the exact gradient of the loss function on the whole training set $D = \{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^{N}$
  - Q: Why not?
  - Conceptual reason: gradient descent is guaranteed to lead to the closest local minimum (if η small enough)
  - Practical reason: we cannot fit all training examples into memory (GPU VRAM) at once
- **Stochastic gradient descent** (SGD) – compute the loss, gradients, and update the parameters based on a single training instance
  - Repeat for all training instances
  - Order of instances random (hence the name stochastic)
  - Many parameter updates – slow training

Mini-Batch Gradient Descent

- Mini-batch GD: sweet spot between full GD and SGD
  - We train in the so-called (mini-)batches of B examples (e.g., B = 32)
  - Iteratively (mini-batch after mini-batch):
    1. Select B training examples from the training set D
    2. Compute the loss L_B and gradient $\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta})$ based on B (using the backpropagation algorithm)
    3. Update the parameters $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})$
  - MBGD – more resilient to local minima than GD and faster than SGD
- Training epoch: model updated on all mini-batches B from D,
  - Each training example part of exactly one mini-batch
  - It is common to train neural LMs for multiple epochs

Mini-batch gradient descent

- Mini-batch GD: sweet spot between full GD and SGD
  - We train in the so-called mini-batches of B examples (e.g., B = 32)
  - MBSG – more resilient to local minima than GD and faster than SGD

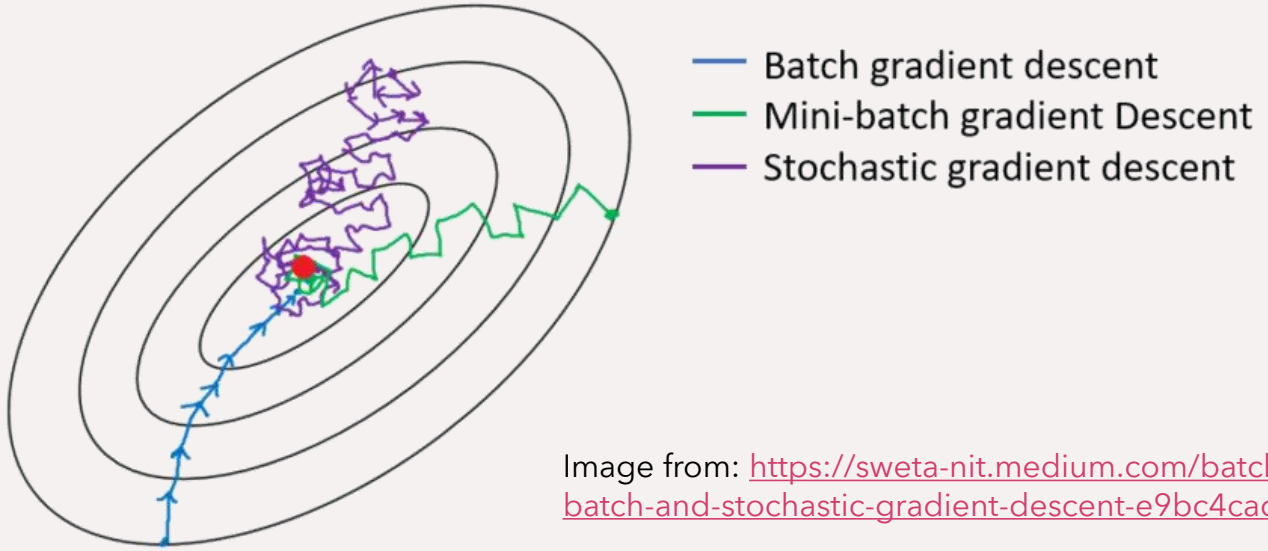


Image from: https://sweta-nit.medium.com/batch-mini-batch-and-stochastic-gradient-descent-e9bc4cacd461

Content

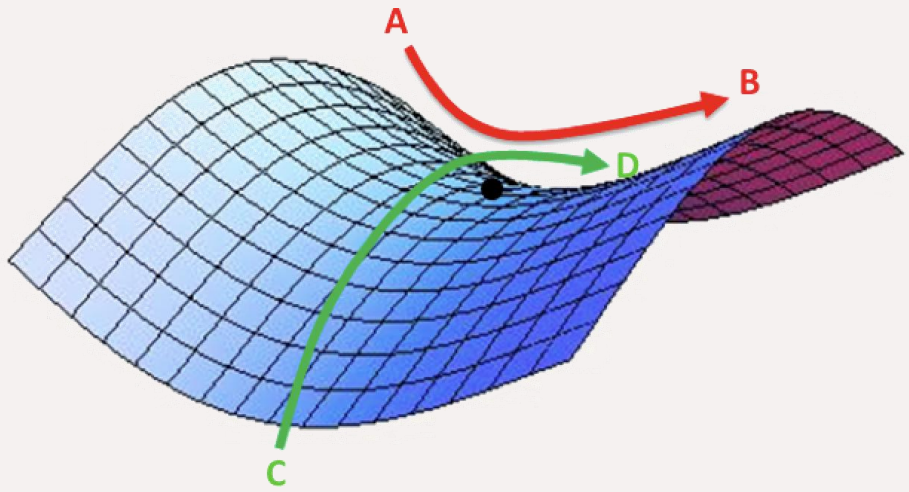
- Uniformity of NLP with Neural LMs
- Training Neural LMs
  - Gradient Descent & Backpropagation
- **Adaptive Optimization**
  - **Momentum, AdaGrad, RMSProp, Adam**
- Dropout

Adapted GD Algorithms

- Let t (time-step) be the counter of the updates to model's parameters
  - t = 1 → first update of parameters, based on gradient of first mini-batch

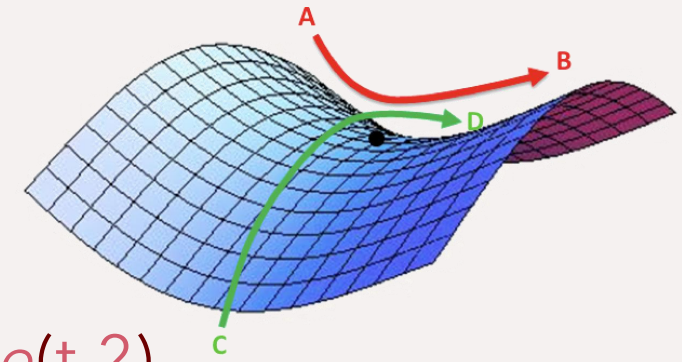
- Mini-batch GD: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta(t)\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})$
  - Update size determined with learning rate, $\eta(t)$, and the gradient $\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})$

  - Problem in saddle points
    - Gradient is zero or close to zero
    - Learning effectively stops



Gradient Descent with Momentum

- To avoid this „stopping", adaptations of GD keep information about the momentum, i.e., previous sizes of parameter changes
  - GD: $change(t) = \eta(t)\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})$,
    $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - change(t)$
  - GD with Momentum:
    - $change(t) = \beta * \eta(t)\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)}) + (1-\beta) * change(t-1)$
    - $change(t-1) = \beta * \eta(t-1)\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t-1)}) + (1-\beta) * change(t-2)$
    - ...
    - $change(t = 1) = \eta(1)\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(0)})$



  - Exponentially weighted averages of current and past updates
    - β is the hyperparameter of the momentum algorithm

Adaptive Gradient (AdaGrad)

- GD makes the step of the same size η in all directions (for all parameters)
  - But the gradient $\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})$ is not of the same size in all directions
  - Optimum is not equally distant from the current point in all dimensions

- Q: A separate learning rate η_i for each parameter $\boldsymbol{\theta}_i$?
  - Not feasible for neural LMs (100M+ to 1T parameters)

- AdaGrad: adaptively scales the learning rate for each parameter – the scaling factor is the sum of the sizes of the gradient squares across all updates

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \frac{\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})}{\sum_{i=1}^{t} \nabla^2_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(i)})}$$

- The size of the update to each parameter depends on the size of the current gradient with respect to the sum of all gradients up to now

Root Mean Square Propagation (RMSProp)

- The sum of the squares of all previous gradients in AdaGrad quickly becomes much larger than any current gradient
  - Updates become small, and optimization slow

- **RMSProp**: introduces a decay on the sum of gradient squares
- $g(t) = \nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})$ – gradient at time step t
- $s(t)$ = sum of gradient squares with decay at time step t

- $s(t) = \beta * s(t-1) + (1-\beta) * g^2(t)$
- $s(1) = g^2(1)$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \frac{g(t)}{s(t)}$$

Adaptive Moment Estimation (Adam)

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. International Conference on Learning Representations (ICLR).

- **Adam** combines **momentum** and **RMSProp** (squared momentum)
  - Empirically shown to work very well in practice
  - The most common choice for optimization of neural LMs (cited over 200K times!)
- $g(t) = \nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})$ – gradient at time step t
- $s_1(t)$ = sum of past gradients with decay at time step t
  - $s_1(t) = \beta_1 * s_1(t-1) + (1-\beta_1)*g(t)$
- $s_2(t)$ = sum of past gradient squares with decay at time step t
  - $s_2(t) = \beta_2 * s_2(t-1) + (1-\beta_2)*g^2(t)$
- $s_1(1) = g(1)$
- $s_2(1) = g^2(1)$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \frac{s_1(t)}{s_2(t)}$$

Content

- Uniformity of NLP with Neural LMs
- Training Neural LMs
  - Gradient Descent & Backpropagation
- Adaptive Optimization
  - Momentum, AdaGrad, RMSProp, Adam
- **Dropout**

Dropout

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of Machine Learning Research, 15(1), 1929-1958..

- Motivation: the risk of model's **overfitting** is related to the ratio of:
  - Number of model's parameters
  - Number of training examples
- If the number of parameters is much larger than the number of training examples, the model will likely overfit to the training data
  - Will not generalize well
- Neural LMs have a lot of parameters

Dropout

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of Machine Learning Research, 15(1), 1929-1958..

- Regularization by training multiple models (multiple instances of deep NNs) and ensembling their predictions is effective
  - But this is very computationally prohibitive!
  - Especially if models are LLMs with billions of parameters ☺

- **Dropout**: a regularization method that simulates training many (slightly) different models in a single training procedure
  - By means of randomly **dropping out** "neurons"
  - Applied on per-layer basis, i.e., on the output of a layer

Dropout

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of Machine Learning Research, 15(1), 1929-1958..

- Let **x** be any hidden representation, output of any layer (e.g., in our neural LM)
  - E.g., output of layer K
- Applying dropout on a layer means
  - To modify its output(s) **x** so that each element x_i becomes replaced with x'_i:

    $x'_i = 0$ with dropout probability p or
    $x'_i = x_i / (1\text{-}p)$ with the probability $(1\text{-}p)$

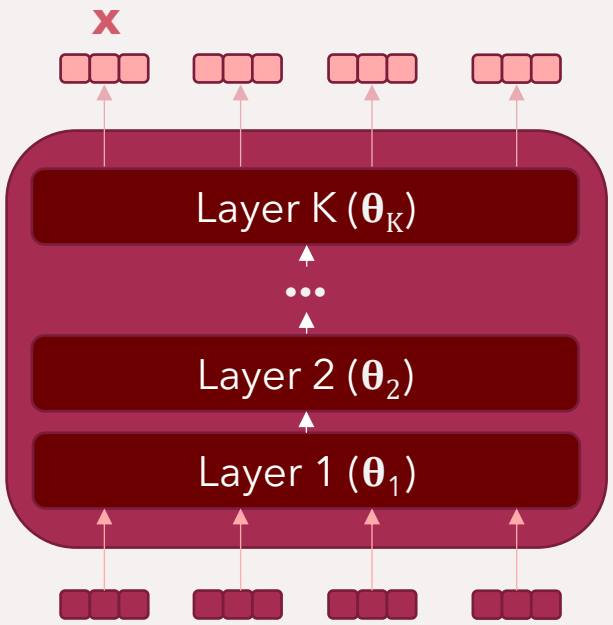


Image: Alexander Mikhalchyk