TECHNISCHE UNIVERSITÄT DARMSTADT

Deep Learning for NLP
Lecture 3: Training as Optimization and (Neural) Language Models

Dr. Mohsen Mesgar

Ubiquitous Knowledge Processing Lab (UKP Lab)

This lecture

- training as optimization
- backpropagation
- language modeling

Recall

- the input to a supervised learning algorithm is a training set $(x_{1:n}, y_{1:n})$, where

Recall

- the input to a supervised learning algorithm is a training set $(x_{1:n}, y_{1:n})$, where
  - $x_{1:n} = x_1, x_2, ..., x_n$ shows input examples

Recall

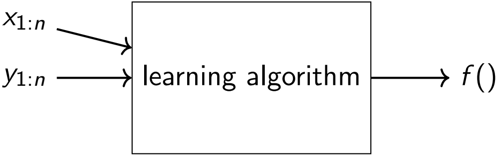
- the input to a supervised learning algorithm is a training set $(x_{1:n}, y_{1:n})$, where
  - $x_{1:n} = x_1, x_2, ..., x_n$ shows input examples
  - $y_{a:n} = y_1, y_2, ..., y_n$ shows corresponding labels

Recall

- the input to a supervised learning algorithm is a training set $(x_{1:n}, y_{1:n})$, where
  - $x_{1:n} = x_1, x_2, ..., x_n$ shows input examples
  - $y_{a:n} = y_1, y_2, ..., y_n$ shows corresponding labels
- the goal of a learning algorithm is to return a function $f()$ that accurately maps input examples to their desired labels

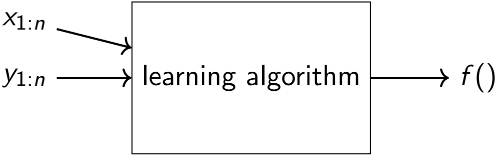
Recall

- the input to a supervised learning algorithm is a training set $(x_{1:n}, y_{1:n})$, where
  - $x_{1:n} = x_1, x_2, ..., x_n$ shows input examples
  - $y_{a:n} = y_1, y_2, ..., y_n$ shows corresponding labels
- the goal of a learning algorithm is to return a function $f()$ that accurately maps input examples to their desired labels



Recall

- the input to a supervised learning algorithm is a training set $(x_{1:n}, y_{1:n})$, where
  - $x_{1:n} = x_1, x_2, ..., x_n$ shows input examples
  - $y_{a:n} = y_1, y_2, ..., y_n$ shows corresponding labels
- the goal of a learning algorithm is to return a function $f()$ that accurately maps input examples to their desired labels



- how to measure if $f()$ works accurately?

Loss Function

- *Loss function* $L(y, \hat{y})$: It quantifies the loss suffered when predicting ŷ while the true label is y

Loss Function

- *Loss function* $L(y, \hat{y})$: It quantifies the loss suffered when predicting ŷ while the true label is y
- given a labeled training set $(x_{1:n}; y_{1:n})$, a per-instance loss function L and a parameterized function $f(x; \Theta)$, we define the corpus-wide loss with respect to the parameters as the average loss over all training examples:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}, y_i),$$

$$\hat{y} = f(x_i; \Theta)$$

Loss Function

- *Loss function* $L(y, \hat{y})$: It quantifies the loss suffered when predicting ŷ while the true label is y
- given a labeled training set $(x_{1:n}; y_{1:n})$, a per-instance loss function L and a parameterized function $f(x; \Theta)$, we define the corpus-wide loss with respect to the parameters as the average loss over all training examples:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}, y_i),$$

$$\hat{y} = f(x_i; \Theta)$$

- in this view the training examples are fixed and the values of the parameters determine the loss

Training as Optimization

- the goal of the training algorithm is then to set the values of the parameters such that the value of $\mathcal{L}$ is minimized

$$\hat{\Theta} = \text{argmin}_{\Theta}\mathcal{L}(\Theta) = \text{argmin}_{\Theta}\frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y_i)$$

Common Loss Functions

- Hinge (binary)
  - for binary classification problems

Common Loss Functions

- Hinge (binary)
  - for binary classification problems
  - the classifier's output is a single scalar $\hat{y}$ and the true output y is in $\{-1, +1\}$

Common Loss Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

- Hinge (binary)
  - for binary classification problems
  - the classifier's output is a single scalar $\hat{y}$ and the true output y is in $\{-1, +1\}$
  - the inference rule is prediction$=$ sign($\hat{y}$), and a classification is considered correct if $y \cdot \tilde{y} > 0$

Common Loss Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

- Hinge (binary)
  - for binary classification problems
  - the classifier's output is a single scalar $\hat{y}$ and the true output y is in $\{-1, +1\}$
  - the inference rule is prediction$=$ sign($\hat{y}$), and a classification is considered correct if $y \cdot \tilde{y} > 0$
  - per-instance loss:

$$L_{\text{hinge(binary)}}(\hat{y}, y) = \max(0, 1 - y.\hat{y})$$

Common Loss Functions

- Hinge (multi-class)
  - let $\hat{y} = \hat{y}_{[1]}, \hat{y}_{[2]}, ..., \hat{y}_{[n]}$ be the model's output vector, and y be the one-hot vector for the correct output class

Common Loss Functions

- Hinge (multi-class)
  - let $\hat{y} = \hat{y}_{[1]}, \hat{y}_{[2]}, ..., \hat{y}_{[n]}$ be the model's output vector, and y be the one-hot vector for the correct output class
  - the inference rule is defined as selecting the class with the highest score prediction $= \text{argmax}_i \hat{y}_{[i]}$

Common Loss Functions

- Hinge (multi-class)
  - let $\hat{y} = \hat{y}_{[1]}, \hat{y}_{[2]}, ..., \hat{y}_{[n]}$ be the model's output vector, and y be the one-hot vector for the correct output class
  - the inference rule is defined as selecting the class with the highest score prediction $= \text{argmax}_i \hat{y}_{[i]}$
  - if t is the correct class and k is the highest scoring class such that $k \neq t$ then loss is

$$L_{\text{hinge(multiclass)}}(\hat{y}, y) = \max(0, 1 - (\hat{y}_{[t]} - \hat{y}_{[k]}))$$

Common Loss Functions

- Log loss
  - can be seen as a "soft" version of the hinge loss with an infinite margin

Common Loss Functions

- Log loss
  - can be seen as a "soft" version of the hinge loss with an infinite margin
  - loss

$$L_{\text{log}}(\hat{y}, y) = \log(1 + \exp(-(\hat{y}_{[t]} - \hat{y}_{[k]}))$$

Common Loss Functions

- binary cross entropy (logistic loss)

Common Loss Functions

- binary cross entropy (logistic loss)
  - used for binary classification with conditional probability outputs.

Common Loss Functions

- binary cross entropy (logistic loss)
  - used for binary classification with conditional probability outputs.
  - assumes a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$

Common Loss Functions

- binary cross entropy (logistic loss)
  - used for binary classification with conditional probability outputs.
  - assumes a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$
  - the model's output $\tilde{y}$ is transformed using the sigmoid function

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

Common Loss Functions

- binary cross entropy (logistic loss)
  - used for binary classification with conditional probability outputs.
  - assumes a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$
  - the model's output $\tilde{y}$ is transformed using the sigmoid function

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

  - then $\hat{y} = \sigma(\tilde{y}) = P(y = 1|x)$

Common Loss Functions

- binary cross entropy (logistic loss)
  - used for binary classification with conditional probability outputs.
  - assumes a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$
  - the model's output $\tilde{y}$ is transformed using the sigmoid function

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

  - then $\hat{y} = \sigma(\tilde{y}) = P(y = 1|x)$
  - inference rule: prediction$= 0$ if $\hat{y} < 0.5$ and prediction$= 1$ if $\hat{y} >= 0.5$

Common Loss Functions

- binary cross entropy (logistic loss)
  - used for binary classification with conditional probability outputs.
  - assumes a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$
  - the model's output $\tilde{y}$ is transformed using the sigmoid function

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

  - then $\hat{y} = \sigma(\tilde{y}) = P(y = 1|x)$
  - inference rule: prediction$= 0$ if $\hat{y} < 0.5$ and prediction$= 1$ if $\hat{y} >= 0.5$
  - loss

$$L_{\text{logistic}}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) log(1 - \hat{y})$$

Common Loss Functions

- binary cross entropy (logistic loss)
  - used for binary classification with conditional probability outputs.
  - assumes a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$
  - the model's output $\tilde{y}$ is transformed using the sigmoid function

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

  - then $\hat{y} = \sigma(\tilde{y}) = P(y = 1|x)$
  - inference rule: prediction$= 0$ if $\hat{y} < 0.5$ and prediction$= 1$ if $\hat{y} >= 0.5$
  - loss

$$L_{\text{logistic}}(\hat{y}, y) = -y\log(\hat{y}) - (1 - y)log(1 - \hat{y})$$

  - is useful for estimating class conditional probability for a binary classification problem

Common Loss Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

- binary cross entropy (logistic loss)
  - used for binary classification with conditional probability outputs.
  - assumes a set of two target classes labeled 0 and 1, with a correct label $y \in \{0, 1\}$
  - the model's output $\tilde{y}$ is transformed using the sigmoid function

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

  - then $\hat{y} = \sigma(\tilde{y}) = P(y = 1|x)$
  - inference rule: prediction$= 0$ if $\hat{y} < 0.5$ and prediction$= 1$ if $\hat{y} >= 0.5$
  - loss

$$L_{\text{logistic}}(\hat{y}, y) = -y\log(\hat{y}) - (1 - y)log(1 - \hat{y})$$

  - is useful for estimating class conditional probability for a binary classification problem
  - we assume that the output layer is transformed using sigmoid

Training as Optimization

- the goal of the training algorithm is then to set the values of the parameters such that the value of $\mathcal{L}$ is minimized

$$\hat{\Theta} = \mathrm{argmin}_{\Theta}\mathcal{L}(\Theta) = \mathrm{argmin}_{\Theta}\frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y_i)$$

Training as Optimization

- the goal of the training algorithm is then to set the values of the parameters such that the value of $\mathcal{L}$ is minimized

$$\hat{\Theta} = \text{argmin}_{\Theta}\mathcal{L}(\Theta) = \text{argmin}_{\Theta}\frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y_i)$$

- the above optimization attempts to minimize the loss at all costs, which may result in overfitting the training data

Regularization

- to counter that we define a regularization term $R(\Theta)$ taking as input the parameters and returning a scalar that reflect their "complexity," which we want to keep low

Regularization

- to counter that we define a regularization term $R(\Theta)$ taking as input the parameters and returning a scalar that reflect their "complexity," which we want to keep low
- training

$$\hat{\Theta} = \mathrm{argmin}_{\Theta}\mathcal{L}(\Theta) = \mathrm{argmin}_{\Theta}\frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y_i)$$

Regularization

- to counter that we define a regularization term $R(\Theta)$ taking as input the parameters and returning a scalar that reflect their "complexity," which we want to keep low
- training

$$\hat{\Theta} = \mathrm{argmin}_{\Theta}\mathcal{L}(\Theta) = \mathrm{argmin}_{\Theta}\frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y_i)$$

- training with regularization

$$\hat{\Theta} = \mathrm{argmin}_{\Theta}\mathcal{L}(\Theta) = \mathrm{argmin}_{\Theta}\left(\frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y_i) + \lambda R(\Theta)\right)$$

Regularization

- intuitively we would like to drive the learner toward natural solutions, in which it is OK to mis-classify a few examples if they don't fit well with the rest

$$\hat{\Theta} = \text{argmin}_{\Theta} \left(\frac{1}{n} \sum_{i=1}^{n} L(\hat{y}, y_i) + \lambda R(\Theta) \right)$$

Regularization

- intuitively we would like to drive the learner toward natural solutions, in which it is OK to mis-classify a few examples if they don't fit well with the rest

$$\hat{\Theta} = \text{argmin}_{\Theta} \left(\frac{1}{n} \sum_{i=1}^{n} L(\hat{y}, y_i) + \lambda R(\Theta) \right)$$

- regularization term considers the parameter values, and scores their complexity

Regularization

- intuitively we would like to drive the learner toward natural solutions, in which it is OK to mis-classify a few examples if they don't fit well with the rest

$$\hat{\Theta} = \text{argmin}_{\Theta} \left(\frac{1}{n} \sum_{i=1}^{n} L(\hat{y}, y_i) + \lambda R(\Theta) \right)$$

- regularization term considers the parameter values, and scores their complexity
- in practice the regularizers equate complexity with large weights and work to keep the parameter values low

Common Regularization Functions

- L_2 regularization (a.k.a. gaussian prior or weight decay): It keeps the sum of the squares of the parameter values low

$$R_{L_2}(W) = ||W||_2^2 = \sum_{i,j} (W_{[i,j]})^2$$

Common Regularization Functions

- L_2 regularization (a.k.a. gaussian prior or weight decay): It keeps the sum of the squares of the parameter values low

$$R_{L_2}(W) = ||W||_2^2 = \sum_{i,j} (W_{[i,j]})^2$$

- the learner will prefer to decrease the value of one parameter with high weight by 1 than to decrease the value of ten parameters that already have relatively low weights by 0.1 each

Common Regularization Functions

- L_1 regularization (a.k.a. sparse prior or lasso): It keeps the sum of the absolute values of the parameters low

$$R_{L_1}(W) = ||W||_1 = \sum_{i,j} |W_{[i,j]}|$$

Common Regularization Functions

- L_1 regularization (a.k.a. sparse prior or lasso): It keeps the sum of the absolute values of the parameters low

$$R_{L_1}(W) = ||W||_1 = \sum_{i,j} |W_{[i,j]}|$$

- the learner will prefer to decrease all the non-zero parameter values toward zero

Common Regularization Functions

- Elastic-Net: combines both L_1 and L_2 regularization

$$R_{\text{elastic-net}}(W) = \gamma_1 R_{L_1}(W) + \gamma_2 R_{L_2}(W)$$

Common Regularization Functions

- Elastic-Net: combines both L_1 and L_2 regularization

$$R_{\text{elastic-net}}(W) = \gamma_1 R_{L_1}(W) + \gamma_2 R_{L_2}(W)$$

- dropout: will be discussed later

Training as Optimization

- we learned that the goal of training is to minimize a loss function (and a regularization term)

$$\hat{\Theta} = \mathrm{argmin}_{\Theta} \left(\mathcal{L}(\hat{y}, y) + \lambda R(\Theta) \right)$$

Training as Optimization

- we learned that the goal of training is to minimize a loss function (and a regularization term)

$$\hat{\Theta} = \text{argmin}_{\Theta} \left(\mathcal{L}(\hat{y}, y) + \lambda R(\Theta)\right)$$

- training = Solving an optimization problem

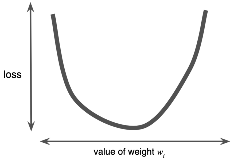
Training as Optimization

- we learned that the goal of training is to minimize a loss function (and a regularization term)

$$\hat{\Theta} = \mathrm{argmin}_{\Theta} \left(\mathcal{L}(\hat{y}, y) + \lambda R(\Theta) \right)$$

- training = Solving an optimization problem
- how to find parameter values that minimize loss?

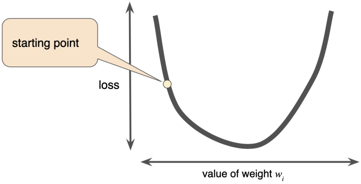
Gradient-based Optimization



(Taken from:

https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

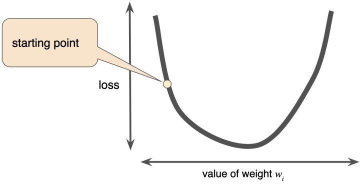
Gradient-based Optimization



(Taken from:

https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

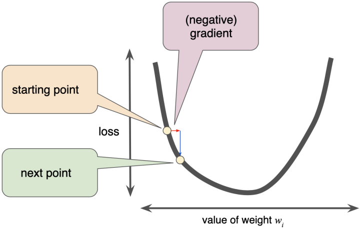
Gradient-based Optimization



(Taken from:

https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

Gradient-based Optimization



(Taken from:

https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

Gradient-based Optimization

TECHNISCHE UNIVERSITÄT DARMSTADT

- we repeatedly compute an estimate of the loss over the training set

Gradient-based Optimization

- we repeatedly compute an estimate of the loss over the training set
- we compute the gradients of the parameters with respect to the loss estimate

Gradient-based Optimization

- we repeatedly compute an estimate of the loss over the training set
- we compute the gradients of the parameters with respect to the loss estimate
- we move the parameter values in the opposite directions of the gradient

(Online) Stochastic Gradient Descent

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x};\Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1},\ldots,\boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1},\ldots,\boldsymbol{y_n}$.
- Loss function L.

while stopping criteria not met **do**
    Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
    Compute the loss $L(f(\boldsymbol{x_i};\Theta), \boldsymbol{y_i})$
    $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i};\Theta), \boldsymbol{y_i})$ w.r.t Θ
    $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
return Θ

(Taken from: Neural Network Methods for Natural Language Processing, Yoav Goldberg)

(Minibatch) Stochastic Gradient Descent

TECHNISCHE UNIVERSITÄT DARMSTADT

Algorithm 2.2 Minibatch stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x};\Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1},\dots,\boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1},\dots,\boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2:     Sample a minibatch of m examples $\{(\boldsymbol{x_1},\boldsymbol{y_1}),\dots,(\boldsymbol{x_m},\boldsymbol{y_m})\}$
3:     $\hat{\boldsymbol{g}} \leftarrow 0$
4:     **for** $i = 1$ to m **do**
5:         Compute the loss $L(f(\boldsymbol{x_i};\Theta),\boldsymbol{y_i})$
6:         $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} +$ gradients of $\frac{1}{m}L(f(\boldsymbol{x_i};\Theta),\boldsymbol{y_i})$ w.r.t Θ
7:     $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
8: **return** Θ

(Taken from: Neural Network Methods for Natural Language Processing, Yoav Goldberg)

Why Minibatch SGD?

- it's not expensive: while computing loss function gradient on all training set can be computationally expensive

Why Minibatch SGD?

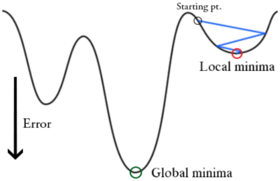
- it's not expensive: while computing loss function gradient on all training set can be computationally expensive
- it converges faster to a good solution than full-batch learning, in which we use all training set to compute gradient

Why Minibatch SGD?

- it's not expensive: while computing loss function gradient on all training set can be computationally expensive
- it converges faster to a good solution than full-batch learning, in which we use all training set to compute gradient
- smaller mini-batch sizes lead often to better solutions (generalize better)

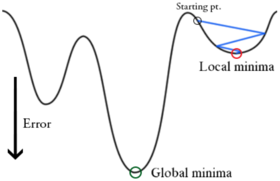
Stochastic Gradient Descent (SGD)

- gradient descent does not always lead to best solutions



Stochastic Gradient Descent (SGD)

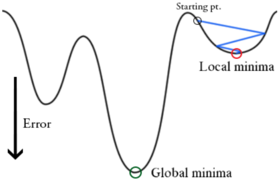
- gradient descent does not always lead to best solutions



- why?

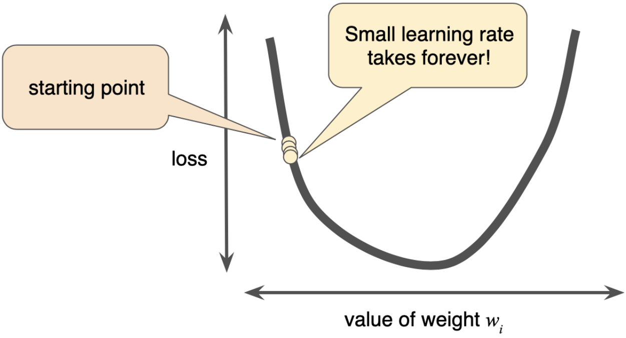
Stochastic Gradient Descent (SGD)

- gradient descent does not always lead to best solutions



- why?
- SGD is sensitive to the learning rate and initial parameter values (starting point)

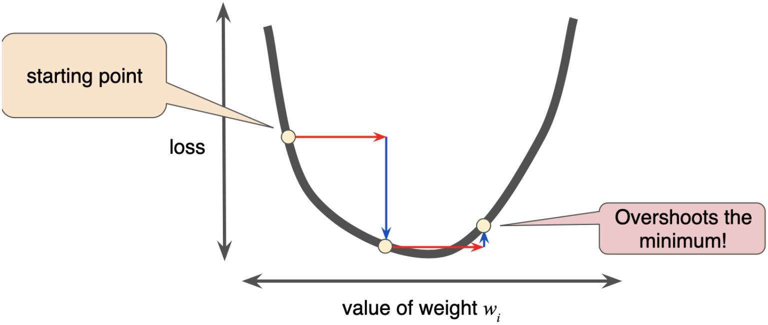
Small Learning Rate



(Taken from:

https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

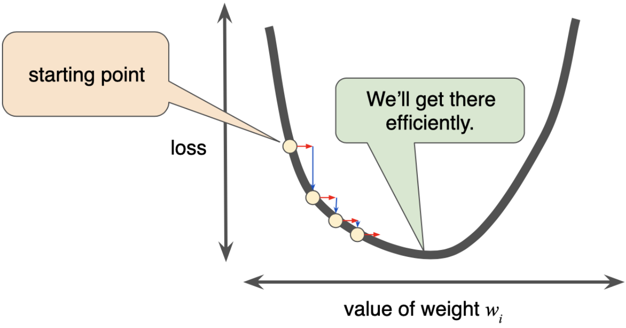
Large Learning Rate



(Taken from:

https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

Adaptive Learning Rate



(Taken from:

https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

Stochastic Gradient Descent (SGD)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Use adaptive learning rate algorithms
  - AdaGrad [Duchi et al., 2011],
  - AdaDelta [Zeiler, 2012],
  - RMSProp [Tieleman and Hinton, 2012],
  - Adam [Kingma and Ba, 2014]

Stochastic Gradient Descent (SGD)

- Use adaptive learning rate algorithms
  - AdaGrad [Duchi et al., 2011],
  - AdaDelta [Zeiler, 2012],
  - RMSProp [Tieleman and Hinton, 2012],
  - Adam [Kingma and Ba, 2014]
- these methods train usually faster than SGD

Stochastic Gradient Descent (SGD)

- Use adaptive learning rate algorithms
  - AdaGrad [Duchi et al., 2011],
  - AdaDelta [Zeiler, 2012],
  - RMSProp [Tieleman and Hinton, 2012],
  - Adam [Kingma and Ba, 2014]
- these methods train usually faster than SGD
- found solution is often not as good as that by SGD $\rightarrow$ First train with Adam, fine-tune with SGD

Stochastic Gradient Descent (SGD)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Use adaptive learning rate algorithms
  - AdaGrad [Duchi et al., 2011],
  - AdaDelta [Zeiler, 2012],
  - RMSProp [Tieleman and Hinton, 2012],
  - Adam [Kingma and Ba, 2014]
- these methods train usually faster than SGD
- found solution is often not as good as that by SGD $\rightarrow$ First train with Adam, fine-tune with SGD
- they use different initial parameter values in different runs of experiments and report the average of scores

Backpropagation

- a fancy name for a recursive algorithm that computes the derivatives of a nested functions using the chain rule, while caching intermediary derivatives

Backpropagation

- a fancy name for a recursive algorithm that computes the derivatives of a nested functions using the chain rule, while caching intermediary derivatives
- chain rule: Assume $y = f(g(x))$

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g} \times \frac{\partial g}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$$

Backpropagation

- consists of two steps

Backpropagation

TECHNISCHE UNIVERSITÄT DARMSTADT

- consists of two steps
  - forward pass $\rightarrow$ use current parameter values to compute the loss value

Backpropagation

- consists of two steps
  - forward pass $\rightarrow$ use current parameter values to compute the loss value
  - backward pass $\rightarrow$ use the gradient of the loss to update the parameter values

Backpropagation: Forward Pass

$$h^{(1)}$$

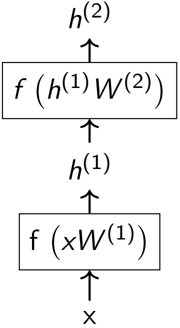
$$\uparrow$$

$$\boxed{\mathrm{f}\left(xW^{(1)}\right)}$$

$$\uparrow$$

$$\mathrm{x}$$

Backpropagation: Forward Pass



Backpropagation: Forward Pass

$$\hat{y}$$

$$\uparrow$$

$$\boxed{f\left(h^{(2)} W^{(3)}\right)}$$

$$\uparrow$$

$$h^{(2)}$$

$$\uparrow$$

$$\boxed{f\left(h^{(1)} W^{(2)}\right)}$$

$$\uparrow$$

$$h^{(1)}$$

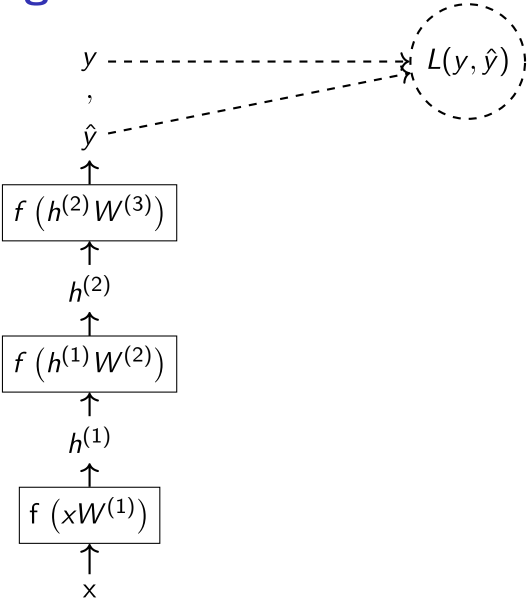
$$\uparrow$$

$$\boxed{\mathrm{f}\left(xW^{(1)}\right)}$$

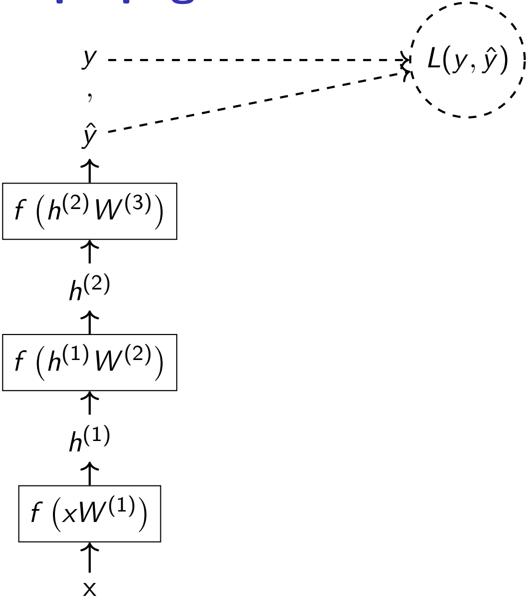
$$\uparrow$$

$$\mathrm{x}$$

Backpropagation: Forward Pass

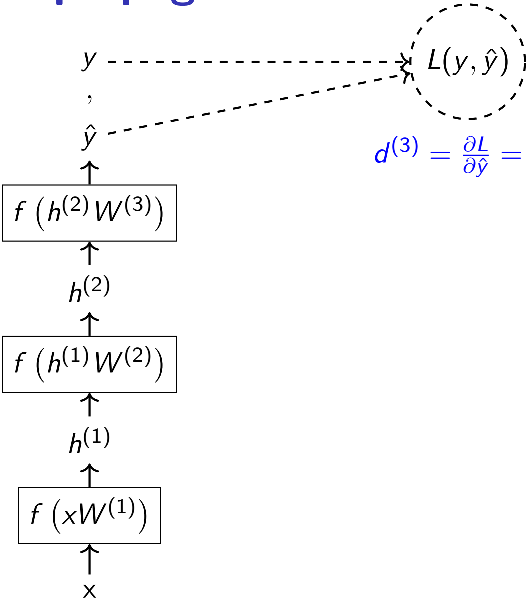


Backpropagation: Backward Pass



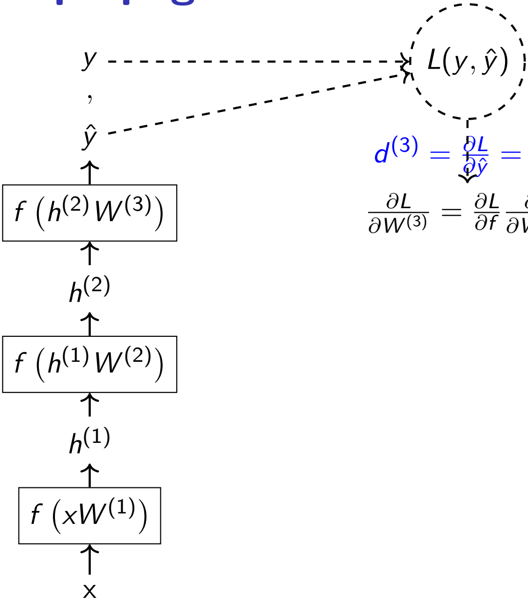
Backpropagation: Backward Pass

TECHNISCHE UNIVERSITÄT DARMSTADT



$$d^{(3)} = \frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial f}$$

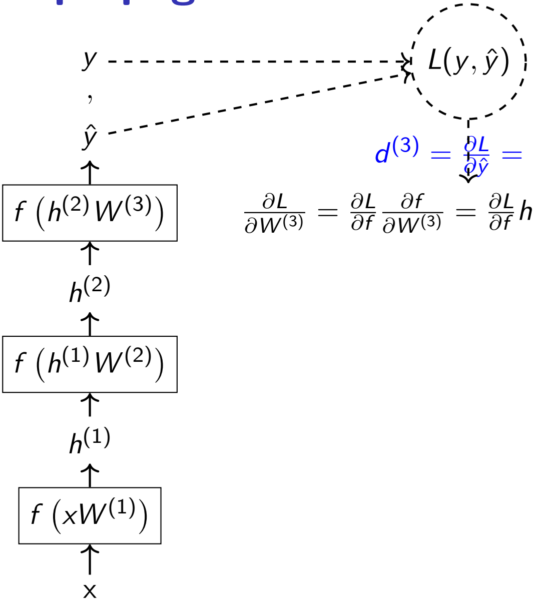
Backpropagation: Backward Pass



$$d^{(3)} = \frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial f}$$

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial W^{(3)}}$$

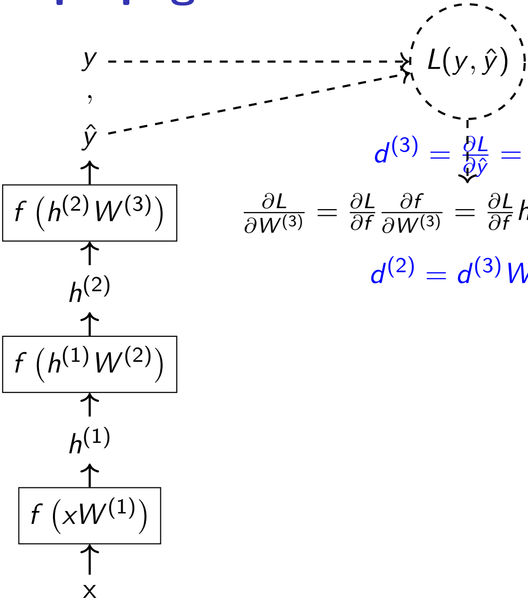
Backpropagation: Backward Pass



$$d^{(3)} = \frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial f}$$

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial W^{(3)}} = \frac{\partial L}{\partial f} h^{(2)} = d^{(3)} h^{(2)}$$

Backpropagation: Backward Pass

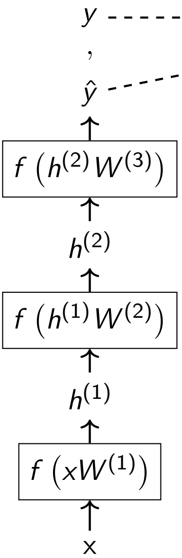


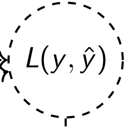
$$d^{(3)} = \frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial f}$$

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial W^{(3)}} = \frac{\partial L}{\partial f}h^{(2)} = d^{(3)}h^{(2)}$$

$$d^{(2)} = d^{(3)}W^{(3)}$$

Backpropagation: Backward Pass





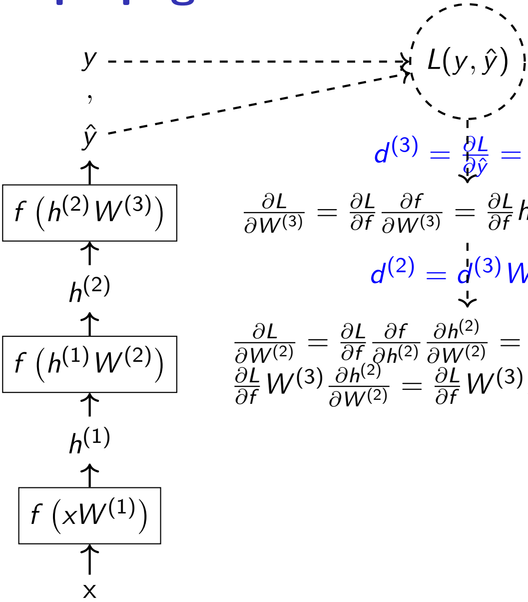
$$d^{(3)} = \frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial f}$$

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial W^{(3)}} = \frac{\partial L}{\partial f}h^{(2)} = d^{(3)}h^{(2)}$$

$$d^{(2)} = d^{(3)}W^{(3)}$$

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial W^{(2)}} =$$

Backpropagation: Backward Pass



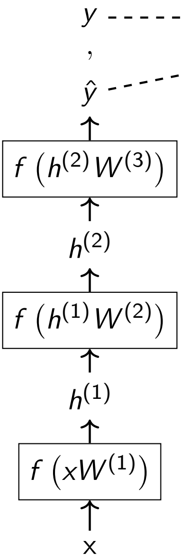
$$d^{(3)} = \frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial f}$$

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial W^{(3)}} = \frac{\partial L}{\partial f}h^{(2)} = d^{(3)}h^{(2)}$$

$$d^{(2)} = d^{(3)}W^{(3)}$$

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial W^{(2)}} =$$
$$\frac{\partial L}{\partial f}W^{(3)}\frac{\partial h^{(2)}}{\partial W^{(2)}} = \frac{\partial L}{\partial f}W^{(3)}h^{(1)} = d^{(2)}h^{(1)}$$

Backpropagation: Backward Pass



$L(y, \hat{y})$

$$d^{(3)} = \frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial f}$$

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial W^{(3)}} = \frac{\partial L}{\partial f}h^{(2)} = d^{(3)}h^{(2)}$$

$$d^{(2)} = d^{(3)}W^{(3)}$$

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial h^{(2)}}\frac{\partial h^{(2)}}{\partial W^{(2)}} =$$
$$\frac{\partial L}{\partial f}W^{(3)}\frac{\partial h^{(2)}}{\partial W^{(2)}} = \frac{\partial L}{\partial f}W^{(3)}h^{(1)} = d^{(2)}h^{(1)}$$

$$d^{(1)} = d^{(2)}W^{(2)}$$

Backpropagation: Backward Pass

y, $\hat{y}$ → $L(y, \hat{y})$

$$\boxed{f\left(h^{(2)} W^{(3)}\right)} \rightarrow \hat{y}$$

$$h^{(2)}: \boxed{f\left(h^{(1)} W^{(2)}\right)}$$

$$h^{(1)}: \boxed{f\left(x W^{(1)}\right)}$$

$$x$$

$$d^{(3)} = \frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial f}$$

$$\frac{\partial L}{\partial W^{(3)}} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial W^{(3)}} = \frac{\partial L}{\partial f} h^{(2)} = d^{(3)} h^{(2)}$$

$$d^{(2)} = d^{(3)} W^{(3)}$$

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial W^{(2)}} = \frac{\partial L}{\partial f} W^{(3)} \frac{\partial h^{(2)}}{\partial W^{(2)}} = \frac{\partial L}{\partial f} W^{(3)} h^{(1)} = d^{(2)} h^{(1)}$$

$$d^{(1)} = d^{(2)} W^{(2)}$$

$$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial W^{(1)}} = \frac{\partial L}{\partial f} W^{(3)} W^{(2)} x = d^{(1)} x$$

Backprop + SGD

TECHNISCHE UNIVERSITÄT DARMSTADT

- the output of backprop is gradient of parameters of a neural model

Backprop + SGD

- the output of backprop is gradient of parameters of a neural model
- once we have the gradients we can use SGD rule to update the parameter values

A Simple Training Loop in PyTorch

```
optimizer = SGD(model_params, lr)

for epoch in range(num_epochs):
    for x,y in  data_batches:

        y_hat = model(x)
        loss = loss_func(y_hat, y)

        optimizer.zero_grad()
        loss.backward()

        optimizer.step()
```

Language Models (LMs)

TECHNISCHE UNIVERSITÄT DARMSTADT

- language modeling is the task of assigning a probability to a sentence in a language

Language Models (LMs)

- language modeling is the task of assigning a probability to a sentence in a language
- what is the probability of seeing the sentence "The cat sat on the mat."

Language Models (LMs)

TECHNISCHE UNIVERSITÄT DARMSTADT

- language modeling is the task of assigning a probability to a sentence in a language
- what is the probability of seeing the sentence "The cat sat on the mat."
- ideal performance at language modeling is to predict the next token in a sequence with a number of guesses that is the identical to or lower than the number of guesses required by a human expert

Language Models (LMs)

- language modeling is the task of assigning a probability to a sentence in a language
- what is the probability of seeing the sentence "The cat sat on the mat."
- ideal performance at language modeling is to predict the next token in a sequence with a number of guesses that is the identical to or lower than the number of guesses required by a human expert
- even without achieving human-level performance, language modeling is a crucial component in real-world NLP applications such as conversational AI, machine-translation, text summarization, ...

Language Models (LMs)

- assume a sequence of words $w_{1:n} = w_1 w_2 ... w_{n-1} w_n$

Language Models (LMs)

- assume a sequence of words $w_{1:n} = w_1 w_2 ... w_{n-1} w_n$
- $P(w_{1:n}) =$
  $P(w_1)P(w_2|w_1)P(w_3|w_{1:2})P(w_4|w_{1:3})...P(w_n|w_{1:n-1})$

Language Models (LMs)

- assume a sequence of words $w_{1:n} = w_1 w_2 ... w_{n-1} w_n$
- $P(w_{1:n}) =$
  $P(w_1)P(w_2|w_1)P(w_3|w_{1:2})P(w_4|w_{1:3})...P(w_n|w_{1:n-1})$
- each word is predicted conditioned on the preceding words

Language Models (LMs)

- assume a sequence of words $w_{1:n} = w_1 w_2 ... w_{n-1} w_n$
- $P(w_{1:n}) =$
  $P(w_1)P(w_2|w_1)P(w_3|w_{1:2})P(w_4|w_{1:3})...P(w_n|w_{1:n-1})$
- each word is predicted conditioned on the preceding words
- estimating the probability of the last token needs to be conditioned on $n-1$ preceding words which is computationally expensive

Language Models (LMs)

- assume a sequence of words $w_{1:n} = w_1 w_2 ... w_{n-1} w_n$
- $P(w_{1:n}) =$
  $P(w_1)P(w_2|w_1)P(w_3|w_{1:2})P(w_4|w_{1:3})...P(w_n|w_{1:n-1})$
- each word is predicted conditioned on the preceding words
- estimating the probability of the last token needs to be conditioned on $n-1$ preceding words which is computationally expensive
- markov-assumption: the future is independent of the past given the present

Language Models (LMs)

- kth order markov-assumption assumes that the next word in a sequence depends only on the last k words

$$P(w_i|w_{1:i-1}) \approx P(w_i|w_{(i-1)-k:i-1})$$

Language Models (LMs)

- kth order markov-assumption assumes that the next word in a sequence depends only on the last k words

$$P(w_i|w_{1:i-1}) \approx P(w_i|w_{(i-1)-k:i-1})$$

- probability of a sequence of tokens $w_{1:n}$

$$P(w_{1:n}) \approx \prod_{i=1}^{n} P(w_i|w_{i-k:i-1})$$

Language Models (LMs)

- kth order markov-assumption assumes that the next word in a sequence depends only on the last k words

$$P(w_i|w_{1:i-1}) \approx P(w_i|w_{(i-1)-k:i-1})$$

- probability of a sequence of tokens $w_{1:n}$

$$P(w_{1:n}) \approx \prod_{i=1}^{n} P(w_i|w_{i-k:i-1})$$

- to make it computationally-friendly for computers (What is the problem with the above format?)

$$\log_2 P(w_{1:n}) \approx \sum_{i=1}^{n} \log_2 \left(P(w_i|w_{i-k:i-1})\right)$$

Evaluating LMs

- the perplexity metric over an unseen sentence indicates how well a LM predicts the likelihood of the sentence

$$\text{prep}_{w_{1:n}}(\text{LM}) = 2^{-\frac{1}{n}\sum_{i=1}^{n}\log_2 \text{LM}(w_i|w_{1:i-1})}$$

Evaluating LMs

- the perplexity metric over an unseen sentence indicates how well a LM predicts the likelihood of the sentence

$$\text{prep}_{w_{1:n}}(\text{LM}) = 2^{-\frac{1}{n}\sum_{i=1}^{n}\log_2 \text{LM}(w_i|w_{1:i-1})}$$

- in this way, we can compare several language models with one another

Evaluating LMs

TECHNISCHE UNIVERSITÄT DARMSTADT

- the perplexity metric over an unseen sentence indicates how well a LM predicts the likelihood of the sentence

$$\text{prep}_{w_{1:n}}(\text{LM}) = 2^{-\frac{1}{n}\sum_{i=1}^{n}\log_2 \text{LM}(w_i|w_{1:i-1})}$$

- in this way, we can compare several language models with one another
- low perplexity values indicate a better language model as it assigns high probabilities to the unseen sentences

Evaluating LMs

- the perplexity metric over an unseen sentence indicates how well a LM predicts the likelihood of the sentence

$$\text{prep}_{w_{1:n}}(\text{LM}) = 2^{-\frac{1}{n}\sum_{i=1}^{n}\log_2 \text{LM}(w_i|w_{1:i-1})}$$

- in this way, we can compare several language models with one another
- low perplexity values indicate a better language model as it assigns high probabilities to the unseen sentences
- perplexities of two language models are only comparable with respect to the same evaluation dataset

Estimating Probabilities

- count-based: The estimates can be derived from corpus counts.

Estimating Probabilities

- count-based: The estimates can be derived from corpus counts.
- let $\#(w_{i:j})$ be the count of the sequence of words $w_{i:j}$ in a corpus

Estimating Probabilities

- count-based: The estimates can be derived from corpus counts.
- let $\#(w_{i:j})$ be the count of the sequence of words $w_{i:j}$ in a corpus
- the maximum likelihood estimate (MLE) of $P(w_i|w_{i-1-k:i-1})$

$$P(w_i|w_{i-1-k:i-1}) = \frac{\#(w_{i-1-k:i})}{\#(w_{i-1-k:i-1})}$$

Estimating Probabilities

- count-based: The estimates can be derived from corpus counts.
- let $\#(w_{i:j})$ be the count of the sequence of words $w_{i:j}$ in a corpus
- the maximum likelihood estimate (MLE) of $P(w_i|w_{i-1-k:i-1})$

$$P(w_i|w_{i-1-k:i-1}) = \frac{\#(w_{i-1-k:i})}{\#(w_{i-1-k:i-1})}$$

- example: $w_1 w_2 w_3 =$ the cat sat

$$P(w_3|w_{1:2}) = \frac{\#(\text{the cat sat})}{\#(\text{the cat})}$$

Estimating Probabilities

TECHNISCHE UNIVERSITÄT DARMSTADT

- what if $\#(w_{i:j}) = 0$?

Estimating Probabilities

- what if $\#(w_{i:j}) = 0$?
- then the probability estimation is 0, which translates to an infinite perplexity

Estimating Probabilities

TECHNISCHE UNIVERSITÄT DARMSTADT

- what if $\#(w_{i:j}) = 0$?
- then the probability estimation is 0, which translates to an infinite perplexity
- one way of avoiding zero-probability N-grams is to use smoothing techniques

Estimating Probabilities

- what if $\#(w_{i:j}) = 0$?
- then the probability estimation is 0, which translates to an infinite perplexity
- one way of avoiding zero-probability N-grams is to use smoothing techniques
- additive smoothing: assume $|V|$ is the vocabulary size and $0 < \alpha \leq 1$

$$P(w_i|w_{i-1-k:i-1}) = \frac{\#(w_{i-1-k:i}) + \alpha}{\#(w_{i-1-k:i-1}) + \alpha|V|}$$

Pro and Cons of Discussed LMs

- easy to train, scale to large corpora, and work well in practice

Pro and Cons of Discussed LMs

- easy to train, scale to large corpora, and work well in practice
- scaling to larger N-grams is a problem for MLE-based language models.

Pro and Cons of Discussed LMs

- easy to train, scale to large corpora, and work well in practice
- scaling to larger N-grams is a problem for MLE-based language models.
- the large number of words in the vocabulary means that statistics for larger N-grams will be sparse

Pro and Cons of Discussed LMs

- easy to train, scale to large corpora, and work well in practice
- scaling to larger N-grams is a problem for MLE-based language models.
- the large number of words in the vocabulary means that statistics for larger N-grams will be sparse
- MLE-based language models suffer from lack of generalization across contexts

Pro and Cons of Discussed LMs

- easy to train, scale to large corpora, and work well in practice
- scaling to larger N-grams is a problem for MLE-based language models.
- the large number of words in the vocabulary means that statistics for larger N-grams will be sparse
- MLE-based language models suffer from lack of generalization across contexts
- having observed “black car” and “blue car” does not influence our estimates of the sequence “red car” if we haven’t seen it before

Neural Language Models

- we can use neural models to estimate probabilities for an LM

Neural Language Models

- we can use neural models to estimate probabilities for an LM
- in this way we can overcome the shortcomings of the MLE-based LMs because neural networks

Neural Language Models

- we can use neural models to estimate probabilities for an LM
- in this way we can overcome the shortcomings of the MLE-based LMs because neural networks
  - they allow conditioning on increasingly large context sizes with only a linear increase in the number of parameters

Neural Language Models

TECHNISCHE UNIVERSITÄT DARMSTADT

- we can use neural models to estimate probabilities for an LM
- in this way we can overcome the shortcomings of the MLE-based LMs because neural networks
  - they allow conditioning on increasingly large context sizes with only a linear increase in the number of parameters
  - they support generalization across different contexts

Neural Language Models

- we can use neural models to estimate probabilities for an LM
- in this way we can overcome the shortcomings of the MLE-based LMs because neural networks
  - they allow conditioning on increasingly large context sizes with only a linear increase in the number of parameters
  - they support generalization across different contexts
- we focus on the neural LM that was introduced by Bengio et al. (2003)

Neural Language Models

TECHNISCHE UNIVERSITÄT DARMSTADT

- let $w_{1:k}$ be the given context

Neural Language Models

- let $w_{1:k}$ be the given context
- we want to estimate $P(w_{k+1}|w_{1:k})$

Neural Language Models

- let $w_{1:k}$ be the given context
- we want to estimate $P(w_{k+1}|w_{1:k})$
- we design an MLP neural model, which takes $w_{1:k}$ as input and returns $P(w_{k+1})$ over all words in vocabulary V as output

$$x = [v(w_1), v(w_2), ..., v(w_k)]$$

$$h^{(1)} = g(xW^{(1)} + b^{(1)})$$

$$P(w_{k+1}) = \text{softmax}(h^{(1)}W^{(2)} + b^{(2)})$$

Neural Language Models

- let $w_{1:k}$ be the given context
- we want to estimate $P(w_{k+1}|w_{1:k})$
- we design an MLP neural model, which takes $w_{1:k}$ as input and returns $P(w_{k+1})$ over all words in vocabulary V as output

$$x = [v(w_1), v(w_2), ..., v(w_k)]$$

$$h^{(1)} = g(xW^{(1)} + b^{(1)})$$

$$P(w_{k+1}) = \text{softmax}(h^{(1)}W^{(2)} + b^{(2)})$$

- the training examples are simply word k-grams from the training set, where the identities of the first $k-1$ words are used as features, and the last word is used as the target label for the classification

Neural Language Models

- let $w_{1:k}$ be the given context
- we want to estimate $P(w_{k+1}|w_{1:k})$
- we design an MLP neural model, which takes $w_{1:k}$ as input and returns $P(w_{k+1})$ over all words in vocabulary V as output

$$x = [v(w_1), v(w_2), ..., v(w_k)]$$

$$h^{(1)} = g(xW^{(1)} + b^{(1)})$$

$$P(w_{k+1}) = \text{softmax}(h^{(1)}W^{(2)} + b^{(2)})$$

- the training examples are simply word k-grams from the training set, where the identities of the first $k-1$ words are used as features, and the last word is used as the target label for the classification
- loss function: cross-entropy loss

Neural LMs for Generating Language

- assume that we are given $w_{1:k}$ as context

Neural LMs for Generating Language

- assume that we are given $w_{1:k}$ as context
- we are asked to predict the next word w_{k+1} from vocabulary V

Neural LMs for Generating Language

- assume that we are given $w_{1:k}$ as context
- we are asked to predict the next word w_{k+1} from vocabulary V
- we feed $w_{1:k}$ to our trained MLP-based LM

Neural LMs for Generating Language

- assume that we are given $w_{1:k}$ as context
- we are asked to predict the next word w_{k+1} from vocabulary V
- we feed $w_{1:k}$ to our trained MLP-based LM
- our LM returns $P(w_{k+1})$ of each word in V

Neural LMs for Generating Language

- assume that we are given $w_{1:k}$ as context
- we are asked to predict the next word w_{k+1} from vocabulary V
- we feed $w_{1:k}$ to our trained MLP-based LM
- our LM returns $P(w_{k+1})$ of each word in V
- we pick up the word with the maximum probability to generate the next word

Neural LMs for Generating Language

- assume that we are given $w_{1:k}$ as context
- we are asked to predict the next word w_{k+1} from vocabulary V
- we feed $w_{1:k}$ to our trained MLP-based LM
- our LM returns $P(w_{k+1})$ of each word in V
- we pick up the word with the maximum probability to generate the next word
- we add the the predicted word to the context and repeat the above procedure

Summary

- training as optimization of a loss function
- common loss functions and regularization terms
- gradient descent (GD, SGD): A general technique for optimization
- backprop(agation): An algorithm for deriving gradients in neural models, once gradients are determined we can train a model with SGD
- (Neural) Language Models (LMs)

Thank You!