Master in Data Science

Mining Unsupervised Data
Word Classification

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

FIB

Outline

Outline

Classification setup and notation

- Generally we have a training dataset consisting of samples
  - $\{x_i, y_i\}_{i=1}^{N}$
- x_i are inputs, e.g. words (indices or vectors), sentences, documents, etc
  - Dimension d.
- y_i are labels (one of C classes) we try to predict, for example:
  - classes: sentiment, named entities, buy/sell decision
  - other words
  - later: multi-word sequences

Classification setup and notation (II)

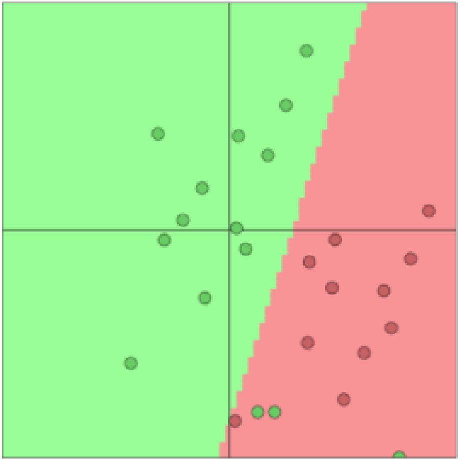


Figure: Simple illustration case: Fixed 2D word vectors to classify. Using softmax/logistic regression. Linear decision boundary.

Outline

Softmax Classifier

- Training Data:
- Traditional ML approach:
  - train (I.e. set) softmax/logistic regression weights $W \in \mathbb{R}^{C \times d}$ to determine a decision boundary (hyperplatne)
- Method: For each x, predict:

$$p(y|x;\theta) = \frac{e^{(W_y \times x)}}{\sum_{c=1}^{C} e^{(W_c \times x)}}$$

Softmax Classifier (II)

$$p(y|x;\theta) = \frac{e^{(W_y \times x)}}{\sum_{c=1}^{C} e^{(W_c \times x)}}$$

We can tease apart the prediction function into two steps:

1. Take the y^{th} row of W and multiply that row with x: $W_y \times x = \sum W_{y_i} x_i{}_{I=1}^{d} = f_y$ Compute all f_c for $c = 1, \ldots, c$
2. Apply softmax function to get the normalised probability:

$$p(y|x;\theta) = \frac{e^{f_y}}{\sum_{c=1}^{C} e^{f_c}} = softmax(f_y)$$

Cross-entropy loss

- For each training example (x, y), our objective is to maximise the probability of the correct class y
- This is equivalent to minimising the negative log probability of that class:

$$-logp(y|x;\theta) = -log(\frac{e^{f_y}}{\sum_{c=1}^{C} e^{f_c}})$$

- Using log probability converts our objective function to sums, which is easier to work with on paper and in implementation.

Cross-entropy loss (II)

- Concept of "cross entropy" is from information theory
- Let the true probability distribution be p
- Let our computed model probability be q
- The cross entropy is:

$$H(p,q) = \sum_{c=1}^{C} p(c) \cdot log(q(c))$$

- Assuming a ground truth (or true or gold or target) probability distribution that is 1 at the right class and 0 everywhere else: $p = [0, \ldots, 0, 1, 0, \ldots 0]$ then:
- Because of one-hot p, the only term left is the negative log probability of the true class

Cross-entropy loss (III)

- Cross entropy loss function over full dataset $x_i, y_i{}_{i=1}^{N}$

$$J(\theta) = \frac{1}{N} \cdot \sum_{i=1}^{N} -log(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}})$$

- In general:

$$\theta = \begin{bmatrix} W_1 \\ \dots \\ W_C \end{bmatrix} = W \in \mathbb{R}^{C \cdot d}$$

- So we only update the decision boundary via:

$$\nabla J(\theta) = \begin{bmatrix} \nabla W_1 \\ \dots \\ \nabla W_C \end{bmatrix} \in \mathbb{R}^{C \cdot d}$$

Outline

Softmax with trainable Word Vectors

- Commonly in NLP deep learning:
  - We learn both W and word vectors x
  - We learn both conventional parameters and representations
  - The word vectors re-represent one-hot vectors (move them around in an intermediate layer vector space) for easy classification with a (linear) softmax classifier

$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla W_1 \\ \dots \\ \nabla W_d \\ \nabla x_{word_1} \\ \dots \\ \nabla x_{word_v} \end{bmatrix} \in \mathbb{R}^{C \cdot d + V \cdot d}$$

! But $V \cdot d$ is big!

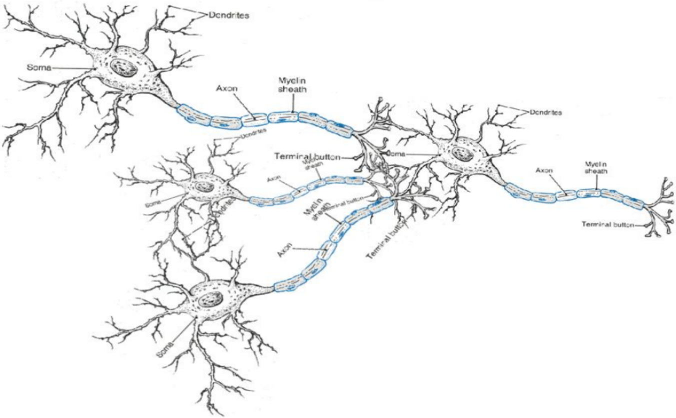
Outline

Neural Network Classifier

- Softmax ($\approx$ logistic regression) alone not very powerful
- Softmax gives only linear decision boundaries This can be quite limiting: Unhelpful when a problem is complex
- Solution: Neural Networks can learn much more complex functions and nonlinear decision boundaries

Figure: Non-linear decision boundary

Neural Computation



A Neuron

A neuron can be a binary logistic regression unit

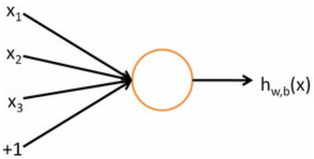
- f = nonlinear activation function (e.g. sigmoid), w = weights, b = bias, h = hidden, x = inputs

$$h_{w,b}(x) = f(w^T \cdot x + b)$$

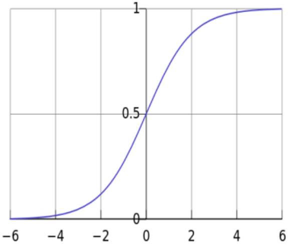
$$f(z) = \frac{1}{1 + e^{-z}}$$

- b = We can have an "always on" feature, which gives a class prior, or separate it out, as a bias term
- w, b are the parameters of this neuron i.e., this logistic regression model

A Neuron



(a) Single Neuron



(b) Sigmoid

Neural Network

- A neural network = running several logistic regressions at the same time
- If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs ...

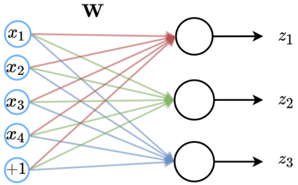


Figure: Neural Network with 3 neurons

But we don't have to decide ahead of time what variables these logistic regressions are trying to predict!

Neural Network (II)

... which we can feed into another logistic regression function

> It is the loss function that will direct what the intermediate hidden variables should be, so as to do a good job at predicting the targets for the next layer, etc.

And if we add more layers... Before we know it, we have a multi-layer neural network....

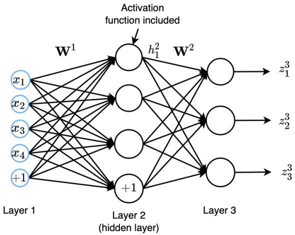


Figure: Multi-layer Neural Network

Neural Network (III)

In a Multi-layer Perceptron (MLP)

$$h(c_1|x;\theta) = \sigma(z) = \frac{1}{1+e^{-z}}$$

z is no longer lineal

$$h(c_k|x;\theta) = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

Then:

$$h_1^2 = f(W_{11}^1 \cdot x_1 + W_{12}^1 \cdot x_2 + W_{13}^1 \cdot x_3 + b_1^1)$$

$$h_2^2 = f(W_{21}^1 \cdot x_1 + W_{22}^1 \cdot x_2 + W_{23}^1 \cdot x_3 + b_2^1)$$

The activation function is applied element-wise

$$f([z_1^2, z_2^2, z_3^2]) = [f(z_1^2), f(z_2^2), f(z_3^2)]$$

The need of Non-linearity

- Without non-linearities, deep neural networks can't do anything more than a linear transform
- Extra layers could just be compiled down into a single linear transform: $W^1 \cdot W^2 \cdot x = W \cdot x$
- More layers approximate more complex functions

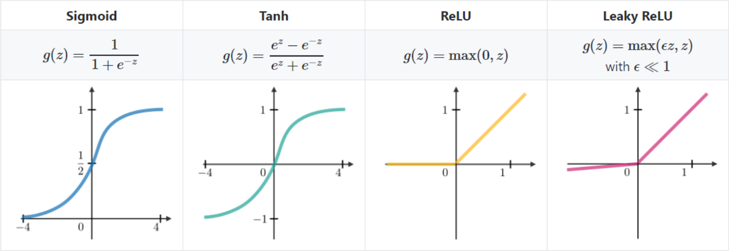


Figure: Common activation functions

You can "play" with them in the TensorFlow Playground

Outline

Outline

Named Entity Recognition - Recap

- We have already introduced the Named Entity Recognition task in the previous session
- Remember that NER aims to find spans of text that are proper names and classify them according to their type: **PER** (person), **LOC** (location), **ORG** (organization), etc.
- We saw that one approach was to use Conditional Random Fields (CRFs)
- Today we will explore neural approaches to NER using word embeddings

Neural Architectures for NER

- Neural architectures for NER typically consist of three main components:
  1. **Word representation layer**: converts words to vectors
  2. **Context encoder**: captures contextual information
  3. **Tag decoder**: assigns entity tags to each word
- Different neural architectures vary in these components
- Today we'll focus on architectures using word embeddings for representation

Outline

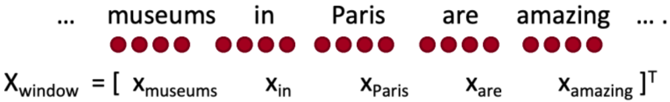
Word-window Classification

- Idea: classify a word in its context window of neighboring words.
  Ex: "Museums in Paris are amazing"
  to classify whether or not the center word "Paris" is a named-entity
- For example, Named Entity Classification of a word in context:
  - Person, Location, Organization, None
- A simple way to classify a word in context might be to average the word vectors in a window and to classify the average vector
  - Problem: that would **lose position information**

Word-window Classification (II)

- Train softmax classifier to classify a center word by taking the concatenation of words surrounding in a window

Ex: Classify "Paris" in the context of this sentence with window length 2:



- Resulting vector $w_{window} = x \in \mathbb{R}^{5 \cdot d}$, a column vector!

Word-window Classification (III)

- With $x = x_{window}$ we can use the softmax classifier

$$p(y|x;\theta) = \frac{e^{z_y}}{\sum_j e^{z_j}} = \frac{e^{W_y \cdot x}}{\sum_j e^{W_j \cdot x}}$$

- With cross-entropy loss:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} -log(\frac{e^{z_{y_i}}}{\sum_{j=1}^{C} e^{z_j}})$$

- How do you update the word vectors?
  - Short answer: Just take derivatives and optimize

Word-window Classification - Binary Logistic Classifier

- Train logistic classifier on hand-labeled data to classify center word {yes / no} for each class based on a concatenation of word vectors in a window

Ex: Classify "Paris" as +/– location in context of sentence with window length 2:

$$X_{window} = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]^T$$

Word-window Classification - Binary Logistic Classifier (II)

- We do supervised training and want high score if it's a location

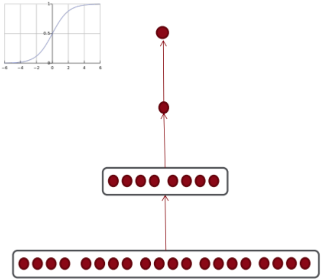
$$J_t(\theta) = \sigma(s) = \frac{1}{1+e^{-s}}$$

predicted model probability of class

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$



Outline

Stochastic Gradient Descent

- Update equation gradient descent:

$$\theta^{new} = \theta^{old} - \alpha \cdot \nabla_\theta J(\theta)$$

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -log(\frac{e^{z_{y_i}}}{\sum_{j=1}^{C} e^{z_j}})$$

- Update equation stochastic gradient descent (SGD):

$$\theta^{new} = \theta^{old} - \alpha \cdot \nabla_\theta J_i(\theta; x_i, y_i)$$

1. Randomly shuffle dataset
2. For every training sample (i) in the dataset-¿apply the update rule

- We can also update the parameter every minibatch, which means a few number of samples.

Gradients

- Given a function with 1 output and n inputs: $f(x) = f(x_1, \mathbf{x}_2, \ldots, x_n)$
- Its gradient is a vector of partial derivatives with respect to each input: $\frac{\partial f}{\partial x} = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}]$
- Now given a function f with m outputs and n inputs, its Jacobian is:

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}) & \ldots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}) & \ldots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}) & \frac{\partial f_m}{\partial x_2}(\mathbf{x}) & \ldots & \frac{\partial f_m}{\partial x_n}(\mathbf{x}) \end{bmatrix}$$

Gradients (II)

- Let's find $\frac{\partial s}{\partial b}$ [1]

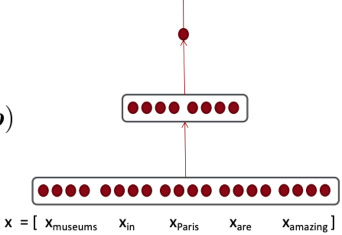
$$J_t(\theta) = \sigma(s) = \frac{1}{1 + e^{-s}}$$

predicted model probability of class

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$



[1] In actuality, we care about the gradient of the loss J_i but we will compute the gradient of the score for simplicity

Gradients (III)

- We apply the chain rule

Ex: Derivative of s respect to b:

$$s = u^T \cdot h \qquad h = f(z) \qquad z = W \cdot x + b$$

$$\frac{\partial s}{\partial b} = \frac{\partial s}{\partial h} \cdot \frac{\partial h}{\partial z} \cdot \frac{\partial z}{\partial b}$$

Computational Graph

- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

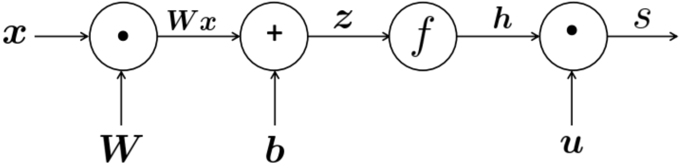


Figure: Forward Pass

Computational Graph (II)

- Then do the backward pass

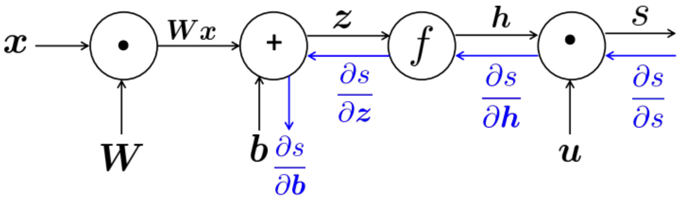
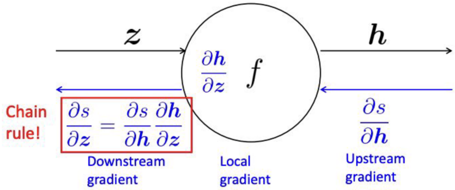


Figure: Backpropagation

Computational Graph (III)

- Backpropagation in a single node:
  - Node receives an “upstream gradient”
  - Goal is to pass on the correct “downstream gradient”
  - Each node has a local gradient
    - The gradient of its output with respect to its input



Outline

Faster Activation Functions

- LReLU (Leaky Retified Linear Unit - Leaky ReLU):
  - Modification ReLU that avoids the "dying ReLU" problem, where neurons stop firing due to a zero output
  - Introduces a small, non-zero slope for negative inputs ($f(x) = \max(\alpha \cdot x, x)$)
  - Can avoid the vanishing gradient problem, which can occur when using sigmoid or other saturating activation functions
  - Allows a small, non-zero gradient when the input is negative, which can prevent the gradient from becoming too small
  - This can lead to faster convergence and better accuracy in some cases.
- ELU (Exponential Linear Unit):
  - Avoids the "dying ReLU" problem and has a smooth output
- SELU (Scaled Exponential Linear Unit):
  - Self-normalizing activation function that can significantly improve the performance

Parameter Initialization

- Proper initialization of model parameters is crucial for effective training and **convergence**. Popular approaches include:
  - Random: e.g., uniform or normal distribution
  - **He**: scaled version of random initialization, designed for ReLU activations
  - **Xavier**: Scaled version of random initialization
    - Designed for sigmoid/tanh activations that have a linear region
    - Sets the variance of the weights to $Var(W_i) = \frac{2}{n_{in}+n_{out}}$, where n_{in} is the number of input neurons and n_{out} is the number of output neurons
  - **Glorot**: Combination of He and Xavier
  - Pre-trained **word-embeddings**: Using pre-trained word-embeddings, such as GloVe or Word2Vec, to initialize the embedding layer of the model
- In general, we initialize the weights to small random values and biases to 0 in the hidden layers.

Optimizers: SGD

- SGD is a commonly used optimizer for neural network training
  - The method iteratively adjusts the model's parameters by computing the **gradient** of the **loss function** with respect to the parameters for a **randomly selected** sample (**stochastic**) of the training data.
- **Simple** and **efficient**.
- However, getting good results often requires **hand-tuning** the **learning rate**
  - Learning rate determines the **step size** that the optimizer takes to update the weights and biases
  - Inappropriate values can cause the optimizer to converge too slowly or too quickly

Adaptative Optimization Algorithms

- They **scale the learning rate** of each parameter based on the accumulated gradient history
- This provides a **per-parameter learning rate** that can perform well in settings with high curvature, noisy gradients, and sparse data
- Popular adaptive optimizers include:
  - Adagrad: divides the learning rate by the sum of the squares of past gradients
  - RMSprop: exponentially decays the average of past squared gradients to normalize the learning rate
  - Adam: combines the benefits of Adagrad and RMSprop by using both first and second moments of past gradients
  - SparseAdam: similar to Adam, but optimized for sparse gradients
- Each optimizer has its own strengths and weaknesses

Learning Rate

- In NLP models, the learning rate plays a crucial role in training and convergence.
- Learning rate determines the size of the step the optimizer takes in the direction of the negative gradient to update the weights and biases of the model.
  - High LR can cause the model to overshoot the optimal point and **diverge**
  - Low LR can result in the model taking too long to converge or getting stuck in **local minima**
- A while a low learning rate can result in the model taking too long to converge or getting stuck in local minima
- NLP models can benefit from using:
  - Learning rate **schedules**
  - **Adaptive** optimization algorithms (previous slide)
- **Fine-tuning pre-trained models** for downstream NLP tasks may require using a smaller learning rate than for training the original model

Regularization

- Regularization (largely) prevents overfitting when we have a lot of features (or later a very powerful/deep model)
- L1 regularization: adds the sum of absolute values of weights to the loss function

$$L_{reg} = L + \lambda \sum_{i=1}^{n} |w_i|$$

- L2 regularization: adds the sum of squares of weights to the loss function

$$L_{reg} = L + \lambda \sum_{i=1}^{n} w_i^2$$

Regularization (II)

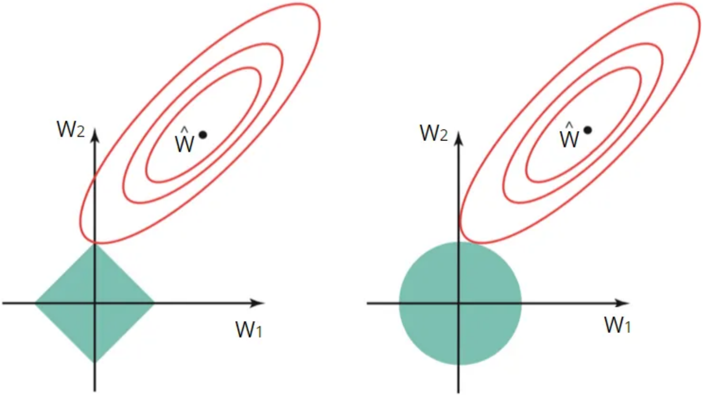
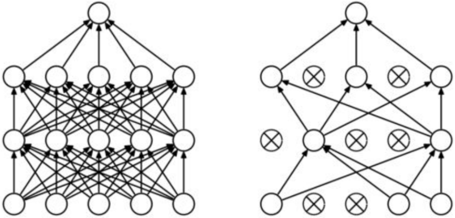


Figure: Representation of the effect of L1 (left) and L2 (right) Regularization. Red lines represent local minima. The red area represents optimal values for the regularization term.

Regularization (III)

- Dropout: randomly sets a fraction of the units to zero during training

$$y = \begin{cases} x & \text{with probability } 1-p \\ 0 & \text{with probability } p \end{cases}$$



Outline

Limitations of Word-window Classification

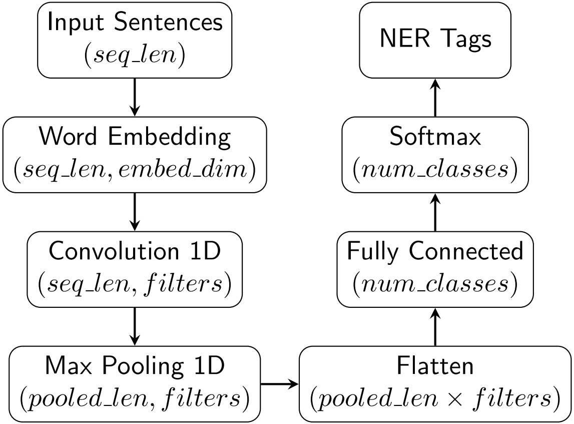
- The sliding window approach has several limitations:
  - **Limited window size**: Fixed context cannot capture long-range dependencies
  - **Local patterns only**: Only captures patterns in the immediate neighborhood of the token
  - **No morphological information**: Cannot leverage subword information (prefixes, suffixes, etc.)
  - **Sparse representations**: Out-of-vocabulary words are problematic
  - **Parameter inefficiency**: Each position in window has separate parameters
- These limitations motivate more sophisticated neural architectures

Outline

Convolutional Neural Networks for NER

- Convolutional Neural Networks (CNNs) can overcome some limitations of the word-window approach
- CNNs apply filters across the input sequence to detect patterns at different positions
- Key benefits:
  - **Parameter sharing**: Same filters applied at different positions
  - **Hierarchical feature extraction**: Stacked CNNs can capture increasingly complex patterns
  - **Position invariance**: Through pooling operations
  - **Variable-length inputs**: Can handle sentences of different lengths

CNN Architecture for NER



CNN for Word Classification

- For a sequence of words $\{w_1, w_2, ..., w_n\}$ with embeddings $\{e_1, e_2, ..., e_n\}$:
  - Apply convolutional filters of width k: $f_j(e_i, e_{i+1}, ..., e_{i+k-1})$
  - Each filter produces a feature map: $c_j = [c_{j,1}, c_{j,2}, ..., c_{j,n-k+1}]$
  - Apply max-pooling over each feature map: $\hat{c}_j = \max(c_j)$
  - Concatenate pooled features from all filters to get a fixed-length representation
  - Feed into fully connected layer and softmax for classification
- This architecture effectively captures local patterns in text

Outline

Convolution Operations for NLP

- Convolution filters extend word windows with several advantages:
  - **Parameter sharing**: Same filter applied across the sequence
  - **Flexible window sizes**: Multiple filter sizes capture different n-gram patterns
  - **Feature detection**: Learn to recognize patterns like negations or entity markers
- Mathematical representation:

$$y_i = f(w_{i:i+n} \cdot \theta + b)$$

where f is typically a non-linear activation function (ReLU)

Pooling Mechanisms

- Pooling operations aggregate filter outputs to:
  - Reduce dimensionality
  - Create position-invariant features
  - Control overfitting
- Max-pooling selects the strongest feature signal:

$$Y_i = \max(y_i, y_{i+1}, ..., y_{i+m})$$

- This operation enables capturing the most salient features regardless of their position

Multi-Layer Convolutions

- Stacked convolutional layers learn hierarchical representations:
  - First layer: Low-level lexical features (character/word sequences)
  - Higher layers: Abstract syntactic/semantic patterns (entity structures)
- Mathematical formulation of stacked layers:

$$Y^1[i] = f(W^1 * X[i : i + k^1] + b^1)$$

$$Y^2[j] = f(W^2 * Y^1[j : j + k^2] + b^2)$$

where $*$ represents convolution operation with stride s

Outline

Character-level Embeddings

- Word embeddings alone face critical limitations:
  - **Out-of-vocabulary words**: Cannot handle unseen words
  - **Missing morphology**: Ignore important subword features
  - **Rare words**: Poor representations for infrequent terms
- Character-level embeddings address these issues by:
  - Representing words as character sequences
  - Learning subword patterns automatically
  - Enabling better modeling of morphologically rich languages
  - Handling unseen words and misspellings gracefully

CNN for Character-level Embeddings

- For each word, represent it as a sequence of character embeddings
- Apply CNN over character sequence for fixed-size word representation:
  - Convolutional layer with multiple filter widths (capturing n-grams)
  - Max-pooling over time to extract salient character patterns
- Character-based CNNs effectively capture:
  - Morphological patterns (prefixes, suffixes)
  - Character-level regularities in named entities

CNN for Character-level Embeddings (II)

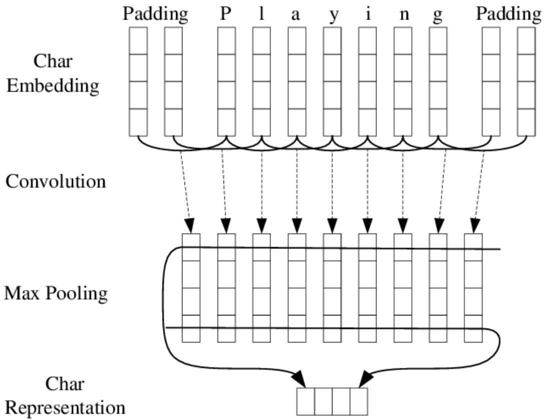


Figure: CNN for Character-level Embeddings

Hybrid Word Representation

- Combining complementary representations improves performance:
  - **Word embeddings**: Capture semantic and distributional information
  - **Character embeddings**: Capture morphological and orthographic patterns
- The concatenated representation provides a more robust word encoding:

$$w_{final} = [w_{pretrained}; w_{char-cnn}]$$

- This hybrid approach effectively handles both seen and unseen words

Word Embeddings + Character Embeddings (II)

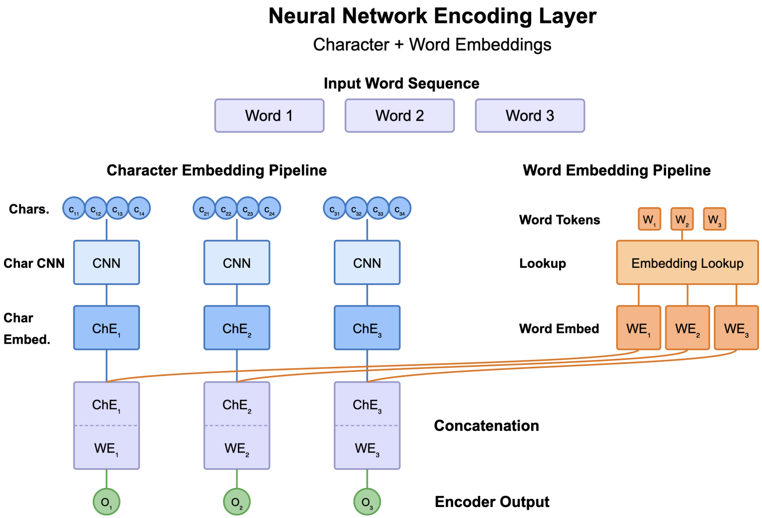


Figure: Hybrid Word Representation: Word + Character Embeddings

Complete Architecture: Embeddings + CNN + MLP + Softmax

- The complete architecture combines:
  1. **Input representation**: Hybrid word+character embeddings
  2. **Feature extraction**: CNN layers for contextual pattern recognition
  3. **Hidden layers**: MLP with non-linear activations
  4. **Output layer**: Softmax for entity type classification
- This architecture effectively addresses the core challenges of NER:
  - Handling unseen entities through character-level patterns
  - Capturing local context through convolution operations
  - Learning non-linear decision boundaries for entity classification

Beyond CNNs: State-of-the-Art Architectures

- CNNs still face certain limitations:
  - Difficulty capturing long-range dependencies
  - Limited modeling of sequential information
- State-of-the-art pre-Transformer architectures combined:
  - **CNN** for character-level features
  - **BiLSTM** for capturing bidirectional context
  - **CRF** layer for modeling label dependencies
- The CNN+BiLSTM+CRF architecture achieved excellent results on NER benchmarks
- BiLSTMs and their integration with CNNs will be covered in future sessions

Outline

Conclusions

- Neural networks excel at NER through their ability to:
  - Capture non-linear patterns in text
  - Learn hierarchical representations from raw data
  - Combine different levels of linguistic information
- Character-level embeddings effectively address OOV and morphological challenges
- Hybrid word+character representations provide robust input for neural architectures
- Modern NER systems benefit from combining CNNs with sequence modeling (BiLSTMs) and structured prediction (CRFs)
- Deep learning for NLP requires careful architecture design and hyperparameter selection