# Convolutional Neural Networks (CNNs)

Deep Learning for NLP: Lecture 6

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München
poerner@cis.uni-muenchen.de

December 09, 2020

## Outline

1. Neural network architectures

2. Convolutional layers
   - Example: 1-D convolution
   - Backpropagation for 1-D convolution
   - Some additional details
   - Convolution with more than one axis
   - Convolution with more than one feature
   - Convolution with a filter bank
   - Assumptions behind convolution

3. Pooling layers

4. CNNs in NLP

## Outline

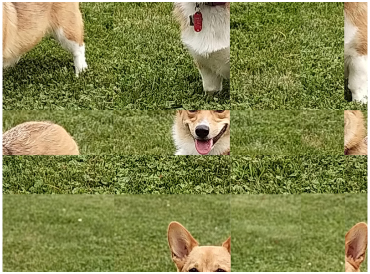
1. Neural network architectures
2. Convolutional layers
   - Example: 1-D convolution
   - Backpropagation for 1-D convolution
   - Some additional details
   - Convolution with more than one axis
   - Convolution with more than one feature
   - Convolution with a filter bank
   - Assumptions behind convolution
3. Pooling layers
4. CNNs in NLP

## What do we mean by "neural network architecture"?

- An architecture is an abstract design for a neural network
- Loosely speaking: Which nodes connect to which nodes? Which connections share parameters?
- Examples of architectures:
  - Fully Connected Neural Networks
  - Convolutional Neural Networks (CNNs)
  - Recurrent Neural Networks (RNNs)
    - Subclasses: Vanilla RNNs, LSTMs, GRUs, QRNNs, ...
  - Transformers (self-attention)
  - ...
- The choice of architecture is often informed by assumptions about the data, based on our domain knowledge

## Assumption of locality

- Computer vision: Pixels that make up a meaningful objects tend to be located in a coherent ("local") area:



- Not always true for words in NLP:
  - *sie sollen sich heute bei ihm im büro* **vorstellen**
  - **stellen** *sie sich bitte heute bei ihm im büro* **vor**

## Assumption of translation invariance

- The features that make up a meaningful object do not depend on that object's absolute position in the input.
- Computer vision:

- NLP:
  - **[the yellow house]** *[rest of sentence]* $\rightarrow$ noun phrase
  - *[rest of sentence]* **[the yellow house]** $\rightarrow$ still a noun phrase

## Assumption of sequentiality

- NLP: Sentences should be processed left-to-right or right-to-left. This one is falling out of fashion, since people are replacing recurrent neural networks with self-attention networks.

## Are these assumptions (always) true?

- Of course not.
- But they are a good way of thinking about why certain architectures are popular for certain problems.
- Also, for limiting the search space when deciding on an architecture for a given project.

## Outline

1 Neural network architectures

2 Convolutional layers
- Example: 1-D convolution
- Backpropagation for 1-D convolution
- Some additional details
- Convolution with more than one axis
- Convolution with more than one feature
- Convolution with a filter bank
- Assumptions behind convolution

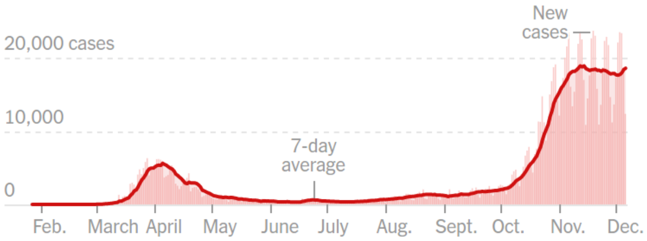
3 Pooling layers

4 CNNs in NLP

## Convolutional layers

- Technique from Computer Vision (esp. object recognition), by LeCun et al., 1998
- Adapted for NLP (e.g., Kalchbrenner et al., 2014)
- Filter banks with trainable filters
- So what is a filter? What is a filter bank?

## Example: 7 day rolling average filter



`https://www.nytimes.com/`, December 07, 2020

## Example: 7 day rolling average filter

- Let $\mathbf{x} \in \mathbb{R}^J$ be a vector that represents a data series (e.g., number of positive tests per day, over $J$ consecutive days)
- Let $\mathbf{f} \in \mathbb{R}^K; \mathbf{f} = [\frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}]^T$ be our "7 day rolling average" filter vector
- Filtering $\mathbf{x}$ with $\mathbf{f}$ (written $\mathbf{f} * \mathbf{x}$) yields a new vector $\mathbf{h}$, with:

$$h_j = \sum_{k=1}^{7} f_k x_{(j+k-7)} = \frac{1}{7}(x_{j-6} + \ldots + x_j)$$

- (For now, let's not worry about the edge cases where $j + k - 7 \leq 0$.)

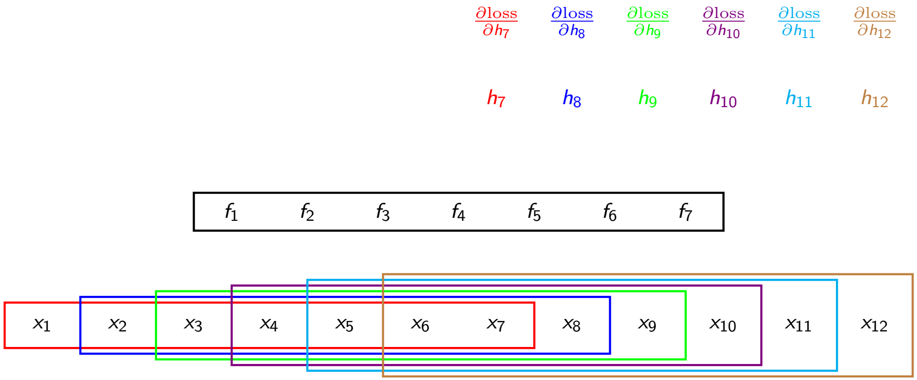
## 1-D convolution

- Let's say that we want to train a linear regression model to predict some variable of interest from each datapoint.
- Since the raw data is noisy, it makes sense to smoothe it with our rolling average filter first:

$$\hat{y}_j = wh_j + b; \quad h_j = (\mathbf{f} * \mathbf{x})_j$$

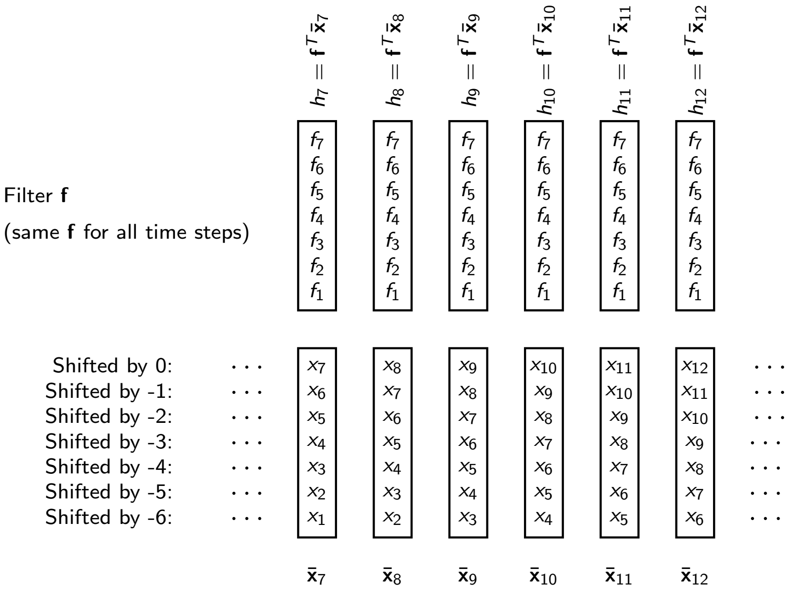
- But maybe we should weight the datapoints in our 7-day window differently? Maybe we should give a higher weight to datapoints close to the current day $j$?
- We can manually engineer a filter that we think is better, for example:
- $\mathbf{f}' = [\frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}]^T$
- Alternative: Let the *model* choose the optimal filter parameters, based on the training data.
- How? Gradient descent!

## Backpropagation for 1-D convolution



$\frac{\partial \text{loss}}{\partial h_7}$ $\frac{\partial \text{loss}}{\partial h_8}$ $\frac{\partial \text{loss}}{\partial h_9}$ $\frac{\partial \text{loss}}{\partial h_{10}}$ $\frac{\partial \text{loss}}{\partial h_{11}}$ $\frac{\partial \text{loss}}{\partial h_{12}}$
$h_7$ $h_8$ $h_9$ $h_{10}$ $h_{11}$ $h_{12}$
$f_1$ $f_2$ $f_3$ $f_4$ $f_5$ $f_6$ $f_7$
$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ $x_9$ $x_{10}$ $x_{11}$ $x_{12}$

# Convolution as dot products



$h_7 = \mathbf{f}^T\bar{\mathbf{x}}_7$
$h_8 = \mathbf{f}^T\bar{\mathbf{x}}_8$
$h_9 = \mathbf{f}^T\bar{\mathbf{x}}_9$
$h_{10} = \mathbf{f}^T\bar{\mathbf{x}}_{10}$
$h_{11} = \mathbf{f}^T\bar{\mathbf{x}}_{11}$
$h_{12} = \mathbf{f}^T\bar{\mathbf{x}}_{12}$
Filter $\mathbf{f}$
(same $\mathbf{f}$ for all time steps)
$f_7$ $f_6$ $f_5$ $f_4$ $f_3$ $f_2$ $f_1$
Shifted by 0: … $x_7$ $x_8$ $x_9$ $x_{10}$ $x_{11}$ $x_{12}$ …
Shifted by -1: … $x_6$ $x_7$ $x_8$ $x_9$ $x_{10}$ $x_{11}$ …
Shifted by -2: … $x_5$ $x_6$ $x_7$ $x_8$ $x_9$ $x_{10}$ …
Shifted by -3: … $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ $x_9$ …
Shifted by -4: … $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ …
Shifted by -5: … $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ …
Shifted by -6: … $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ …
$\bar{\mathbf{x}}_7$ $\bar{\mathbf{x}}_8$ $\bar{\mathbf{x}}_9$ $\bar{\mathbf{x}}_{10}$ $\bar{\mathbf{x}}_{11}$ $\bar{\mathbf{x}}_{12}$

## Backpropagation for 1-D convolution

- If we pretend that there is a separate filter parameter $\mathbf{f}_j$ for each $\bar{\mathbf{x}}_j$, then the gradient of the loss w.r.t. that filter would simply be:

$$\nabla_{\mathbf{f}_j}\text{loss} = \frac{\partial h_j}{\partial \mathbf{f}_j}\frac{\partial \text{loss}}{\partial h_j} = \bar{\mathbf{x}}_j \frac{\partial \text{loss}}{\partial h_j}$$

- But of course, there is only one filter. So we add up the gradients for all time steps:

$$\nabla_{\mathbf{f}}\text{loss} = \sum_j \nabla_{\mathbf{f}_j}\text{loss} = \sum_j \bar{\mathbf{x}}_j \frac{\partial \text{loss}}{\partial h_j}$$

## Questions?

Take a moment to write down any questions you have for the QA session!

## Bias, nonlinearities, padding and stride

- **Bias and nonlinearities:**
  - In a real CNN, we usually add a trainable scalar bias $b$, and we apply a nonlinearity $g$ (such as the rectified linear unit):

$$h_j = g(b + \sum_{k=1}^{K} f_k x_{(j+k-K)})$$

- **Edge cases:**
  - Possibel strategies for when the filter overlaps with the edges (beginning/end) of $\mathbf{x}$:
    - ★ Outputs where filter overlaps with edge are undefined, i.e., $\mathbf{h}$ has fewer dimensions than $\mathbf{x}$ ($K-1$ fewer, to be exact)
    - ★ Pad $\mathbf{x}$ with zeros before applying the filter
    - ★ Pad $\mathbf{x}$ with some other relevant value, such as the overall average, the first/last value, ...
- **Stride:**
  - The stride of a convolutional layer is the "step size" with which the filter is moved over the input
  - We apply $\mathbf{f}$ to the $j$'th window of $\mathbf{x}$ only if $j$ is divisible by the stride.
  - In NLP, the stride is usually 1.

## Convolution with more than one axis

- Extending the convolution operation to tensors with more than one axis is straightforward.
- Let $\mathbf{X} \in \mathbb{R}^{J_1 \times \ldots \times J_L}$ be a tensor that has $L$ axes and let $\mathbf{F} \in \mathbb{R}^{K_1 \times \ldots \times K_L}$ be a filter.
  - The dimensionalities of the filter axes are called filter sizes or kernel sizes ("kernel width", "kernel height", etc.)
  - From now on, we assume that the filter is applied to a symmetric window around position $j$, not just to positions to the left of $j$. (The rolling average was a special scenario, because we assume that future data points $x_{j'}$ with $j' > j$ are not available on day $j$.)
- Then the output $\mathbf{H} = \mathbf{F} * \mathbf{X}$ is a tensor with $L$ axes, where:

$$h_{(j_1,\ldots,j_L)} = g\Big(b + \sum_{k_1=1}^{K_1} \ldots \sum_{k_L=1}^{K_L} f_{(k_1,\ldots,k_L)} x_{(j_1+k_1-\lceil \frac{K_1}{2} \rceil,\ldots,j_L+k_L-\lceil \frac{K_L}{2} \rceil)}\Big)$$

## Example: 2D convolution

Filter $\mathbf{F}$ with $K_2 = K_2 = 3$

$$\boxed{\begin{array}{ccc} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 0 & 1 & 2 \end{array}}$$

Input $\mathbf{X}$ with $J_1 = J_2 = 4$ (padded)

$$\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 2 & 3 & 2 & 0 \\ 0 & 2 & 1 & 2 & 1 & 0 \\ 0 & 2 & 1 & 2 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Output $\mathbf{H}$ with $J_1 = J_2 = 4$
(without bias or nonlinearity)

$$\begin{array}{cccc} 16 & ? & ? & ? \\ ? & ? & 26 & ? \\ ? & ? & ? & ? \\ ? & ? & ? & 5 \end{array}$$

## Channels

- So far, we have assumed that each position in the input (each day or each pixel) contains a single scalar.
- Now, we assume that our input has $M$ features ("channels") per position, i.e., there is an additional feature axis: $\mathbf{X} \in \mathbb{R}^{J_1 \times \ldots \times J_L \times M}$
- Example:
  - $M = 3$ channels per pixel in an RGB (red/green/blue) image
  - In NLP: dimensionality of pretrained word embeddings
- Then our filter gets an additional feature axis, which has the same dimensionality as the feature axis of $\mathbf{X}$:

$$\mathbf{F} \in \mathbb{R}^{K_1 \times \ldots \times K_L \times M}$$

- During convolution, we simply sum over this new axis as well:

$$h_{(j_1,\ldots,j_L)} = g\Big(b + \sum_{k_1=1}^{K_1} \ldots \sum_{k_L=1}^{K_L} \sum_{m=1}^{M} f_{(k_1,\ldots,k_L,m)} x_{(j_1+k_1-\lceil\frac{K_1}{2}\rceil,\ldots,j_L+k_L-\lceil\frac{K_L}{2}\rceil,m)}\Big)$$

## Filter banks

- A filter bank is a tensor that consists of $N$ filters, which each have the same shape.
- The filters of a filter bank are applied to $\mathbf{X}$ independently, and their outputs are stacked (they form a new axis in the output).
- So let this be our input and filter bank:

$$\mathbf{X} \in \mathbb{R}^{J_1 \times \ldots \times J_L \times M}$$

$$\mathbf{F} \in \mathbb{R}^{K_1 \times \ldots \times K_L \times M \times N}$$

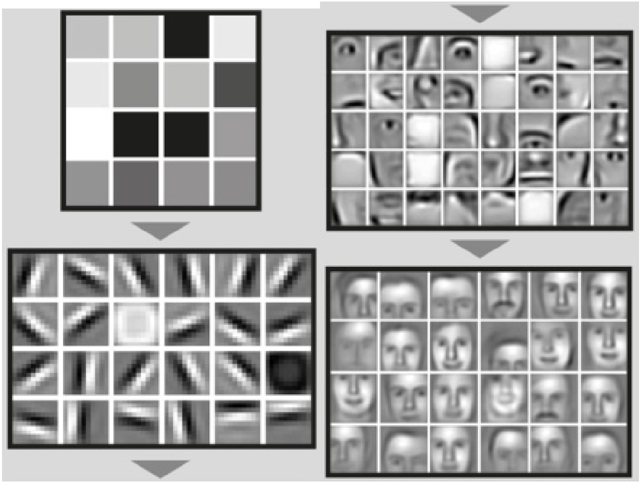
- Then our output is a tensor of shape $\mathbf{H} \in \mathbb{R}^{J_1 \times \ldots \times J_L, N}$
  - (assuming that we are padding the first $L$ axes of $\mathbf{X}$ to preserve their dimensionality in $\mathbf{H}$)
- where:

$$h_{(j_1,\ldots,j_L,n)} = g\big(b + \sum_{k_1=1}^{K_1} \ldots \sum_{k_L=L}^{K_L} \sum_{m=1}^{M} f_{(k_1,\ldots,k_L,m,n)} x_{(j_1+k_1-\left\lceil \frac{K_1}{2} \right\rceil,\ldots,j_L+k_L-\left\lceil \frac{K_L}{2} \right\rceil,m)}\big)$$

## Assumptions behind convolution

- Translation invariance: Same object in different positions.
  - Parameter sharing: Filters are shared between all positions.
- Locality: Meaningful objects form coherent areas
  - In a single convolutional layer, information can travel no further than a few positions (depending on filter size)
- Hierarchy of features from simple to complex:
  - In computer vision, we often apply many convolutional layers one after another
  - With every layer, the information travels further, and the feature vectors become more complex
  - Pixels $\rightarrow$ edges $\rightarrow$ shapes $\rightarrow$ small objects $\rightarrow$ bigger objects

## Convolution



Source: Computer science: The learning machines. Nature (2014).

## Questions?

Take a moment to write down any questions you have for the QA session!

## Outline

1 Neural network architectures

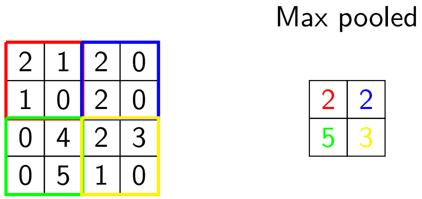
2 Convolutional layers
- Example: 1-D convolution
- Backpropagation for 1-D convolution
- Some additional details
- Convolution with more than one axis
- Convolution with more than one feature
- Convolution with a filter bank
- Assumptions behind convolution

3 Pooling layers

4 CNNs in NLP

## Pooling layers

- Pooling layers are parameter-less
- Divide axes of **H** (excluding the filter/feature axis) into a coarse-grained "grid"
- Aggregate each grid cell with some operator, such as the average or maximum
- Example (shown without a feature axis):

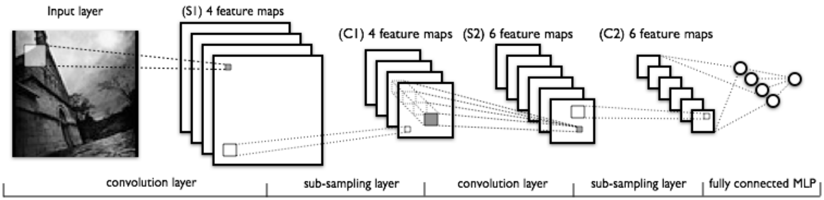


## Pooling layers

- When pooling, we lose fine-grained information about exact positions
- In return, pooling allows us to aggregate features from a local neighborhood into a single feature.
    - For example, if we have a neighborhood where many "dog features" were detected (meaning, shapes that look like parts of a dog), we want to aggregate that information into a single "dog neuron"

## Convolution and Pooling: LeNet

- In computer vision, we often apply pooling between convolutional layers.
- Repeated pooling has the effect of reducing tensor sizes.



LeCun et al. (1998). Gradient-based learning applied to document recognition.

## Outline

1 Neural network architectures

2 Convolutional layers
- Example: 1-D convolution
- Backpropagation for 1-D convolution
- Some additional details
- Convolution with more than one axis
- Convolution with more than one feature
- Convolution with a filter bank
- Assumptions behind convolution

3 Pooling layers

4 CNNs in NLP

## CNNs in NLP

- Images are 2D, but text is a 1D sequence (of words, n-grams, etc).
- Words are usually represented by $M$-dimensional word embeddings (e.g., from Word2Vec)
- So on text, we do 1-D convolution with $M$ input features:
  - Input matrix: $\mathbf{X} \in \mathbb{R}^{J \times M}$
  - Filter bank of $N$ filters: $\mathbf{F} \in \mathbb{R}^{K \times M \times N}$; $K < J$
  - Output matrix: $\mathbf{H} \in \mathbb{R}^{J \times N}$
    - (assuming that we padded $\mathbf{X}$ with zeros along its first axis)
- Usually, CNNs in NLP are not as deep as CNNs in Computer Vision – often just one convolutional layer.

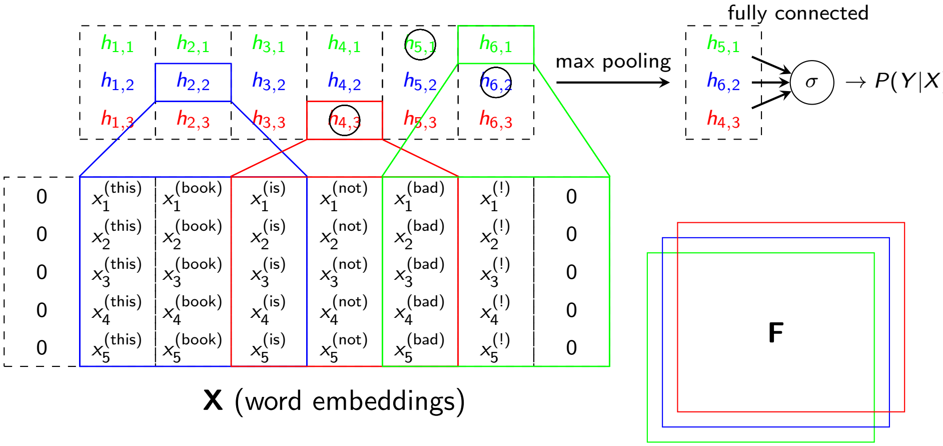
## Pooling in NLP

- Pooling is less frequently used in NLP.
- If the task is a word-level task (i.e., we need one output vector per word), we can simply use the output of the final convolutional layer – no pooling needed.
- If the task is some sentence-level task (i.e., we need a single vector to represent the entire sentence), we usually do a "global" pooling step over the entire sentence *after* the last convolutional layer:

$$\mathbf{h}_{\text{avgpool}} = \frac{1}{J} \sum_{j=1}^{J} \mathbf{h}_j$$

$$\mathbf{h}_{\text{maxpool}} = \begin{bmatrix} \max_j & h_{j,1} \\ & \vdots \\ \max_j & h_{j,N} \end{bmatrix}$$

# CNNs in NLP



fully connected
max pooling
$\sigma$
$\rightarrow P(Y|X$
$h_{1,1}$ $h_{2,1}$ $h_{3,1}$ $h_{4,1}$ $h_{5,1}$ $h_{6,1}$
$h_{1,2}$ $h_{2,2}$ $h_{3,2}$ $h_{4,2}$ $h_{5,2}$ $h_{6,2}$
$h_{1,3}$ $h_{2,3}$ $h_{3,3}$ $h_{4,3}$ $h_{5,3}$ $h_{6,3}$
$h_{5,1}$ $h_{6,2}$ $h_{4,3}$
0 $x_1^{(this)}$ $x_1^{(book)}$ $x_1^{(is)}$ $x_1^{(not)}$ $x_1^{(bad)}$ $x_1^{(!)}$ 0
0 $x_2^{(this)}$ $x_2^{(book)}$ $x_2^{(is)}$ $x_2^{(not)}$ $x_2^{(bad)}$ $x_2^{(!)}$ 0
0 $x_3^{(this)}$ $x_3^{(book)}$ $x_3^{(is)}$ $x_3^{(not)}$ $x_3^{(bad)}$ $x_3^{(!)}$ 0
0 $x_4^{(this)}$ $x_4^{(book)}$ $x_4^{(is)}$ $x_4^{(not)}$ $x_4^{(bad)}$ $x_4^{(!)}$ 0
0 $x_5^{(this)}$ $x_5^{(book)}$ $x_5^{(is)}$ $x_5^{(not)}$ $x_5^{(bad)}$ $x_5^{(!)}$ 0
**X** (word embeddings)
**F**

X = “this book is not bad !”, Y = “positive”

- How many parameters in the filter bank? $3 \times 5 \times 3$ (assuming no bias)

## Questions?

Take a moment to write down any questions you have for the QA session!

# Introduction to pytorch

Deep Learning for NLP: Lecture 6(b)

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München
poerner@cis.uni-muenchen.de

December 09, 2020

## Outline

1. Why pytorch?
2. Tensors and operations
3. Automatic differentiation
4. Modules
5. Loss functions
6. Optimizers
7. Datasets and data loaders
8. Implementing your own model
9. Using pytorch on a GPU
10. Putting it all together

## Outline

1. Why pytorch?
2. Tensors and operations
3. Automatic differentiation
4. Modules
5. Loss functions
6. Optimizers
7. Datasets and data loaders
8. Implementing your own model
9. Using pytorch on a GPU
10. Putting it all together

## Can we implement neural networks with numpy?

- Sure we can, but...
  - ... we would have to implement many functionalities from scratch
  - ... we would have to derive all of our gradients by hand $\rightarrow$ error-prone and annoying
  - ... numpy has no inherent way of grouping functions and their parameters $\rightarrow$ no modularization
  - ... numpy does not support calculations on GPU

## Enter pytorch!

- Open source library for deep learning with Python, developed by Facebook
- Documentation: `http://pytorch.org/docs/master/torch.html`
- 60 minute tutorial: `https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`
- Popular alternatives are *tensorflow* by Google, *CNTK* by Microsoft

## pytorch...

- ... many pre-implemented layers, loss functions and optimizers
- ... can do gradients and backpropagation automatically
- ... object-oriented grouping of functions and parameters into modules
- ... supports calculations on GPU

## Outline

1 Why pytorch?

2 Tensors and operations

3 Automatic differentiation

4 Modules

5 Loss functions

6 Optimizers

7 Datasets and data loaders

8 Implementing your own model

9 Using pytorch on a GPU

10 Putting it all together

## Tensors

- `torch.tensor`: Numerical objects (scalars, vectors, matrices...)
- Every tensor has a data type (dtype) and a shape (which is called *size*)
- Tensors can be manipulated and combined via operations (addition, matrix multiplication, concatenation, etc.)
- The result of a tensor operation is a new tensor
- Think about `torch.tensor` as the pytorch equivalent of `numpy.array` (but with some additional abilities related to gradients)

## Creating tensors

```
>>> import torch
>>> # You can create tensors from nested Python lists:
>>> torch.tensor([[1.0, 1.3], [7.7, 8.0]], dtype=torch.float32)
tensor([[1.0000, 1.3000],
        [7.7000, 8.0000]])
>>> # ... or from numpy arrays:
>>> torch.from_numpy(np.arange(3))
tensor([0, 1, 2])
>>> # ... or from all-zeros / all-ones:
>>> torch.ones(size=(2,3))
>>> torch.zeros(size=(2,3))
>>> # ... or through random initialization:
>>> y = torch.rand(size=(2,), dtype=torch.float32)
>>> # You can easily convert tensors back to numpy arrays:
>>> y.detach().numpy()
array([0.5634323, 0.8529429], dtype=float32)
```

## Tensor operations

- Operations often work similar to numpy (with some exceptions):

```
>>> vector = torch.tensor([1.0, 1.3])
>>> vector2 = torch.tensor([-.5, -.7])
>>> matrix = torch.ones(size=(2,3))
>>> # elementwise addition, subtraction, multiplication, etc.
>>> vector + vector2
>>> vector * vector2
>>> # elementwise unary functions (log, exp, etc.)
>>> torch.log(vector)
>>> torch.exp(vector)
>>> # dot product
>>> vector.dot(vector2)
>>> # matrix multiplication
>>> vector.matmul(matrix)
```

- When you are looking for a specific operation, try googling "pytorch equivalent of numpy ________"

## Outline

1. Why pytorch?
2. Tensors and operations
3. Automatic differentiation
4. Modules
5. Loss functions
6. Optimizers
7. Datasets and data loaders
8. Implementing your own model
9. Using pytorch on a GPU
10. Putting it all together

## Automatic differentiation

$$\begin{tikzcd} \mathbf{w} \arrow[dr] & \\ & \hat{y} = \mathbf{x}^T\mathbf{w} \\ \mathbf{x} \arrow[ur] & \end{tikzcd}$$

$\nabla_{\mathbf{w}}\hat{y} = \mathbf{x}; \qquad \nabla_{\mathbf{x}}\hat{y} = \mathbf{w}$

- When we apply operations to tensors, pytorch implicitly builds a computation graph
- When backpropagation is invoked at a scalar tensor, the gradients of that tensor are computed via automatic differentiation
- The gradients are stored in the `grad` attribute of the input tensors:

```
>>> x = torch.tensor([1., 2., 3.], requires_grad=True) # see note below
>>> w = torch.tensor([0., 0., 1.], requires_grad=True)
>>> y_hat = x.dot(w)
>>> y_hat
tensor(3., grad_fn=<DotBackward>)
>>> y_hat.backward()
>>> (w.grad, x.grad)
(tensor([1., 2., 3.]), tensor([0., 0., 1.]))
```

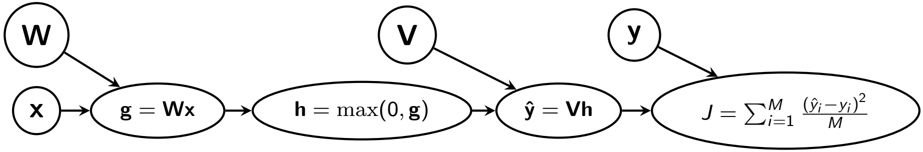
*Note: We are using single-char variable names to highlight the relation with the computation graph. This is not best practice, and you should use more expressive variable names in your own code.

## Disabling automatic differentiation

- Automatic differentiation is expensive:
  - We have to store the content of all intermediate tensors
  - After backpropagation, we have to store the gradients
- Disabling automatic differentiation:
  - Use `requires_grad=False` for tensors that do not require gradients (e.g., “frozen” parameters)
  - When you don’t intend to do gradient updates (e.g., during evaluation), use `torch.no_grad()` context

```
>>> x = torch.tensor([1., 2., 3.], requires_grad=False)
>>> w = torch.tensor([0., 0., 1.], requires_grad=True)
>>> with torch.no_grad():
>>>   y_hat = x.dot(w)
>>>   y_hat.backward()  # no_grad context -> no backpropagation
RuntimeError
>>> y_hat = x.dot(w)
>>> y_hat.backward()
>>> (w.grad, x.grad) # gradients only for tensors with requires_grad=True
(torch.tensor([1., 2., 3.]), None)
```

## Automatic differentiation via chain rule: Feed-Forward Net


$$\begin{tikzcd} \mathbf{W} \arrow[dr] & & & \mathbf{V} \arrow[dr] & & \mathbf{y} \arrow[dr] & \\ \mathbf{x} \arrow[r] & \mathbf{g} = \mathbf{W}\mathbf{x} \arrow[r] & \mathbf{h} = \max(0, \mathbf{g}) \arrow[rr] & & \hat{\mathbf{y}} = \mathbf{V}\mathbf{h} \arrow[rr] & & J = \sum_{i=1}^{M} \frac{(\hat{y}_i - y_i)^2}{M} \end{tikzcd}$$

```
>>> x = torch.tensor([1., 2., 3.], requires_grad=False) # inputs
>>> y = torch.tensor([.7, .8], requires_grad=False) # targets
>>> W = torch.rand(size=(3,2), requires_grad=True) # param 1
>>> V = torch.rand(size=(2,2), requires_grad=True) # param 2
>>> g = x.matmul(W) # hidden vector
>>> h = torch.relu(g) # hidden vector after nonlinearity
>>> y_hat = h.matmul(V) # predicted vector
>>> J = torch.mean((y_hat - y)**2) # MSE loss
>>> J.backward()
>>> W.grad
tensor([[2.1843, 1.4898],
        [4.3687, 2.9796],
        [6.5530, 4.4694]])
>>> V.grad
tensor([[3.7324, 4.8193],
        [7.0224, 9.0673]])
```

## Automatic differentiation

- No matter how complicated our neural network becomes: As long as every function is a (differentiable) pytorch function, pytorch can derive the gradients via the chain rule.
- No more manual backpropagation! 🎉

## Outline

1. Why pytorch?
2. Tensors and operations
3. Automatic differentiation
4. Modules
5. Loss functions
6. Optimizers
7. Datasets and data loaders
8. Implementing your own model
9. Using pytorch on a GPU
10. Putting it all together

## Modules

- Pytorch groups functions and their parameters into *modules* (also called *layers*)
- torch.nn contains many predefined module classes:

```
>>> import torch.nn as nn
>>> nn.Linear(in_features, out_features) # simple linear layer
>>> nn.Conv2d(in_channels, out_channels, kernel_size) # 2D CNN
>>> nn.Embedding(num_embeddings, embedding_dim) # lookup layer
>>> # and many more
```

- Here, we instantiate a linear layer with input size 3 and output size 2:

```
>>> torch.random.manual_seed(0)
>>> linear = nn.Linear(in_features=3, out_features=2, bias=True)
```

## Modules: parameters

- Most modules have parameters
- For example, our linear layer has a parameter matrix and a bias vector
- The parameters are randomly initialized when the module is created
- We can inspect the parameters:

```
>>> list(linear.parameters())
[Parameter containing:
tensor([[-0.0043,  0.3097, -0.4752],
        [-0.4249, -0.2224,  0.1548]], requires_grad=True),
Parameter containing:
tensor([-0.0114,  0.4578], requires_grad=True)]
```

- Note that `requires_grad` is set to `True`, because pytorch expects us to train these parameters.

## Modules: `forward` function

- Every module class has a `forward` function, which handles the forward-pass logic.
- In the case of the linear layer, the forward-pass logic is simply:

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

```
# from https://pytorch.org/docs/stable/_modules/torch
class Linear(Module):
  # (...)
  def forward(self, input: Tensor) -> Tensor:
    return F.linear(input, self.weight, self.bias)

# F.linear:
def linear(input, weight, bias=None):
  # (...)
  output = input.matmul(weight.t())
  if bias is not None:
    output += bias
  ret = output
  return ret
```

## Modules: `forward()` function

- “Calling” a module means calling its forward function:

```
>>> inputs = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
>>> predictions = linear(inputs)
>>> predictions
tensor([[-0.8219,  0.0526],
        [-1.3313, -1.4247]], grad_fn=<AddmmBackward>)
>>> linear.forward(inputs) # this is equivalent
tensor([[-0.8219,  0.0526],
        [-1.3313, -1.4247]], grad_fn=<AddmmBackward>)
```

- Typically, modules expect their inputs to have a batch axis. So in reality, the input is a tensor of shape $(N, \dots)$, where $N$ is the number of datapoints per batch (here: 2), and $\dots$ stands for the remaining axes (here: one 3-dimensional feature axis). The layer is then applied to every datapoint independently.

## Outline

1. Why pytorch?
2. Tensors and operations
3. Automatic differentiation
4. Modules
5. Loss functions
6. Optimizers
7. Datasets and data loaders
8. Implementing your own model
9. Using pytorch on a GPU
10. Putting it all together

## Loss functions

- Many popular loss functions are pre-implemented in `torch.nn`:

```
>>> nn.MSELoss() # mean squared error
>>> nn.BCELoss() # Negative Log Likelihood (binary)
>>> nn.NLLLoss() # Negative Log Likelihood (multi-class)
>>> # and many more
```

- A loss function behaves like a module, except that it has no parameters, and its forward function usually takes two inputs (the prediction and the target):

```
>>> loss_function = nn.MSELoss()
>>> targets = torch.tensor([[.1, .2], [-.3, -.4]])
>>> predictions = linear(inputs)
>>> predictions
tensor([[-0.8219,  0.0526],
        [-1.3313, -1.4247]], grad_fn=<AddmmBackward>)
>>> loss = loss_function(predictions, targets)
>>> loss
tensor(0.7463, grad_fn=<MseLossBackward>)
```

## Backpropagation

- Now we can backpropagate the loss:

```
>>> loss.backward()
```

- Let's look at the gradients of the loss w.r.t. the parameters. There is one gradient tensor per parameter (one for the matrix and one for the bias). These gradients were calculated when we called `loss.backward()`, and they are stored in the `grad` attribute:

```
>>> [param.grad for param in linear.parameters()]
[tensor([[-2.5235, -3.5000, -4.4766],
        [-2.1231, -2.7092, -3.2953]]),
tensor([-0.9766, -0.5861])]
```

## Outline

1 Why pytorch?

2 Tensors and operations

3 Automatic differentiation

4 Modules

5 Loss functions

6 Optimizers

7 Datasets and data loaders

8 Implementing your own model

9 Using pytorch on a GPU

10 Putting it all together

## Gradient update

- Now that we have the gradients, we could update the parameters manually:

```
>>> learning_rate = 0.01
>>> with torch.no_grad():
...   for param in linear.parameters():
...     param.sub_(learning_rate*param.grad)
```

- But that's too much work.
- Instead, we use an optimizer from the `torch.optim` package

## Optimizers

- Optimizers are objects that take care of gradient updates.
- When instantiating an optimizer, you pass the model parameters to it. That way, the optimizer knows what it has to update:

```
>>> import torch.optim as optim
>>> optimizer = optim.SGD(linear.parameters(), lr=0.01)
```

- Pytorch has many complex pre-implemented optimizers
- Complex optimizers can do things like adapt the momentum of the learning rate, use different learning rates for different parameters, etc.

```
>>> optim.Adam(linear.parameters(), lr=0.01)
>>> optim.RMSprop(linear.parameters(), lr=0.01)
>>> # ...and many more. Let's use SGD for now.
```

## Optimizers

- With an optimizer, the gradient update step is simply:

```
>>> optimizer.step() # update all parameters with their gradients
>>> optimizer.zero_grad() # zero out all gradients
```

- Careful: The call to `optimizer.zero_grad()` is easy to forget. But it is important.
  - If we don't zero out gradients, they sum up inside the `grad` tensor. As a result, the "gradient" at training step 1000 would be $\sum_{i=1}^{1000} \nabla_\theta J(\theta; x_i, y_i)$ instead of $\nabla_\theta J(\theta; x_{1000}, y_{1000})$ (where $x_i, y_i$ is the $i$'th training batch)

## Outline

1. Why pytorch?
2. Tensors and operations
3. Automatic differentiation
4. Modules
5. Loss functions
6. Optimizers
7. Datasets and data loaders
8. Implementing your own model
9. Using pytorch on a GPU
10. Putting it all together

## Handling data

- What we could do:

```
>>> all_inputs, all_targets = load_data() # data as tensors
>>> all_inputs.shape, all_targets.shape
(torch.Size([100, 3]), torch.Size([100, 2]))
>>>
>>> custom_shuffling_function(all_inputs, all_targets)
>>> for i in range(0, all_inputs.shape[0], batch_size):
>>>   inputs = all_inputs[i:i+batch_size]
>>>   targets = all_targets[i:i+batch_size]
```

- Runs in same thread as model

## Datasets and Dataloaders

- Datasets store and offer access to data
- DataLoaders take care of iteration, batching, shuffling, ...
- multithreading → we don't waist GPU time waiting for the next batch

```
>>> import torch.utils.data as data
>>> dataset = data.TensorDataset(all_inputs, all_targets)
>>> len(dataset)
100
>>> dataloader = data.DataLoader(dataset, batch_size=3, shuffle=True,
                                 num_workers=4)
>>> len(dataloader) # ceil(len(dataset) / batch_size)
34
>>> for inputs, targets in dataloader:
...  print(inputs)
...  print(targets)
...  break
...
tensor([[0.7601, 0.2101, 0.3622],
        [0.6240, 0.2828, 0.1898],
        [0.7679, 0.9298, 0.4364]])
tensor([[0.4980, 0.3000],
        [0.8256, 0.0253],
        [0.0634, 0.7476]])
```

## Outline

1 Why pytorch?

2 Tensors and operations

3 Automatic differentiation

4 Modules

5 Loss functions

6 Optimizers

7 Datasets and data loaders

8 Implementing your own model

9 Using pytorch on a GPU

10 Putting it all together

## Implementing your own model

- When you implement a model, you will normally define a module class, which inherits from `nn.Module`
- Normally, you will build the model from simpler sub-modules
- To implement the feed-forward network from a few slides ago:

```
>>> import torch.nn.functional as F
>>> class FFN(nn.Module):
...  def __init__(self, input_size, hidden_size, output_size):
...    super(FFN, self).__init__() # mandatory: invoke parent init
...
...    # register sub-modules as attributes
...    self.linear1 = nn.Linear(input_size, hidden_size)
...    self.linear2 = nn.Linear(hidden_size, output_size)
...
...  def forward(self, inputs):
...    hidden_vectors = F.relu(self.linear1(inputs))
...    predictions = self.linear2(hidden_vectors)
...    return predictions
```

## Implementing your own model

- Our model can then be instantiated and “called” like any other module from `torch.nn`:

  ```
  >>> model = FFN(input_size=3, hidden_size=2, output_size=2)
  >>> predictions = model(inputs)
  ```

- The parameters of our model are the parameters of its sub-modules:

```
>>> list(model.parameters()) # 2 matrices, 2 bias vectors
[Parameter containing:
tensor([[ 0.4607, -0.0698,  0.0371],
        [ 0.5576,  0.0167,  0.5723]], requires_grad=True),
Parameter containing:
tensor([0.2204, 0.4270], requires_grad=True),
Parameter containing:
tensor([[-0.1111, -0.6049],
        [ 0.0504,  0.3004]], requires_grad=True),
Parameter containing:
tensor([0.0817, 0.1186], requires_grad=True)]
```

## Outline

1. Why pytorch?
2. Tensors and operations
3. Automatic differentiation
4. Modules
5. Loss functions
6. Optimizers
7. Datasets and data loaders
8. Implementing your own model
9. Using pytorch on a GPU
10. Putting it all together

## Using pytorch on a GPU

- To use pytorch on a GPU, you need a compatible GPU (duh!) and CUDA
  - see `https://pytorch.org/get-started/locally/`
- Alternatively, you can use a cloud GPU via Google Colab (see demo at the end of Lecture 5(b)).
- Note: The coding exercises are designed in such a way that you can solve them on a CPU or GPU. So if you can't make them work on the GPU, don't worry.
- (But if you are thinking about doing a deep learning project for your Master's thesis: This is a good time to get some practice.)

## Using pytorch on a GPU

- First, check if CUDA is available on your machine:

```
>>> torch.cuda.is_available()
True
```

- Then, move your model (your `nn.module` object) to the GPU.

```
>>> model = model.to(device='cuda')
```

- If you have several GPUs, you can select them by their index:

```
>>> model = model.to(device='cuda:2') # indices start at 0
```

- Inputs have to be on the same device as the model:

```
>>> inputs = inputs.to(device='cuda') # or whatever device the model is on
>>> predictions = model(inputs)
```

- Tensors that were computed by your model (here: `predictions`) live on the same device as the model. To move them back to the CPU:

```
>>> predictions.to(device='cpu')
```

## Outline

1 Why pytorch?

2 Tensors and operations

3 Automatic differentiation

4 Modules

5 Loss functions

6 Optimizers

7 Datasets and data loaders

8 Implementing your own model

9 Using pytorch on a GPU

10 Putting it all together

## Putting it all together: The training loop

```
>>> device = 'cuda' if torch.cuda.is_available() else 'cpu'
>>> model = MODELCLASS(...) # e.g., model = FFN(3, 2, 2)
>>> model = model.to(device)
>>> optimizer = OPTIMCLASS(model.parameters(), ...) # e.g., optim.SGD(...)
>>> loss_function = LOSSCLASS(...) # e.g., nn.MSELoss()
>>> X, Y = load_data() # load training data tensors from somewhere
>>> dataset = data.TensorDataset(X, Y)
>>> dataloader = data.DataLoader(dataset, shuffle=True, batch_size=16)
>>>
>>> for epoch in range(10):
>>>   loss_sum = 0.0
>>>   for inputs, targets in dataloader:
>>>     inputs = inputs.to(device)
>>>     targets = targets.to(device)
>>>     predictions = model(inputs)
>>>     loss = loss_function(predictions, targets)
>>>     loss.backward()
>>>     optimizer.step()
>>>     optimizer.zero_grad()
>>>     loss_sum += loss.to('cpu').detach().numpy()
>>>   print('Avg loss for epoch', epoch, ':', loss_sum/len(dataloader))
```