Convolutional Networks for NLP and Complements on Training

Joseph Le Roux

18/08/26

Outline

- Motivation
- Convolutional Networks
- Beyond Convolutions: a Brief History of Sequence Models
- Cross-Entropy Learning
- Practical GD: Learning Rate
- Practical GD: Regularisation and Dropout

1
Motivation

1 NN for classification

Assume we want to classify elements of $\mathbb{R}^d$ into c different classes

With a MLP:

- define a MLP R which computes a function de $\mathbb{R}^d \rightarrow \mathbb{R}^c$
- given element $x \in \mathbb{R}^d$, return $c^* = \arg\max_k R(x)[k]$

Learn parameters of R from labeled examples :

- Let $\mathcal{T} = \{(x_i, k_i)\}_i$ be a set of examples
- maximise its log-likelihood $L = \sum_i \log p(k_i|x_i)$
- softmax to the rescue:

$$p(k|x) = \frac{\exp(R(x)[k])}{\sum_{k'} \exp(R(x)[k'])}$$

- other (non-probabilistic) possibilities...
  - perceptron
  - margin-maximisation (SVM)

1 Sentence Classification

We want to classify sentences/word sequences into different categories, eg:

- tweets from Donald Trump into positive/negative/neutral
- sentences about NLP vs. sentences not about NLP
- During lab session: tweets about public transportation in Île-de-France

  "le temps depuis ce matin il ralenti que quand jdois attendre le bus"

  "@mouloudachour @SNCF ah c' est comme ça vieux. on se fait tous avoir faut vérifier le billet avant de l' acheter."

  "@CollectifLADALA #sncf est-il possible de prendre le 8812 et d´arrivée à l´heure a Montparnasse?"

  "Je suis dans un bus tcl le chauffeur il a crus conduire une Ferrari 😲 #tcl"

1 Variable size input

Until now we have seen how to represent words (*atoms*)

- vectors in $\mathbb{R}^d$, fixed size.
- How to represent sentences?

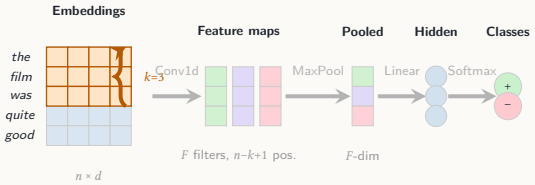
Naïve idea: concatenate vectors

- sentence of n words represented by a vector in $\mathbb{R}^{nd}$
- problem: no unique MLP can transform a vector for a sentence into a vector of class scores, requires one MLP per sentence length.
- but still... concatenation is an important idea

Solution: Convolution networks (with pooling)

- apply a local filter over windows of k words $\longrightarrow$ then pooling compresses to a fixed-size vector
- alternative: recurrent networks (RNNs), discussed briefly at the end of this lecture

1 CNN for Sentence Classification: Full Pipeline



Deep Architecture

- word embeddings non-contextual word representations
- convolutions contextualized representations
- pooling compression on fixed-size representation
- MLP classifier class scores

2
Convolutional Networks

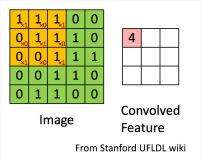
Problem with MLPs: unstructured representation

- all inputs are connected to all outputs (*fully connected* MLP)
- no notion of distance between parts of the inputs (pixels, words...)

Convolution Product

$$(f * g)(n) = \sum_{m \in N_n} f(m) \times g(n), \text{ where } N_n \text{ is a neighbourhood of } n$$

- definition difficult to operate in NNs: usually g is independent of the position
- CNN to compress: g linear mapping into a *smaller* space
  - B&W picture (0/1 grid)
  - yellow: neighbourhood; red : filter
  - pink: output



Convolution example (from C.Manning) (1/5)

A 1D convolution for text

tentative	0.2	0.1	−0.3	0.4
deal	0.5	0.2	−0.3	−0.1
reached	−0.1	−0.3	−0.2	0.4
to	0.3	−0.3	0.1	0.1
keep	0.2	−0.3	0.4	0.2
government	0.1	0.2	−0.1	−0.1
open	−0.4	−0.4	0.2	0.3

t,d,r	−1.0
d,r,t	−0.5
r,t,k	−3.6
t,k,g	−0.2
k,g,o	0.3

Apply a **filter** (or **kernel**) of size 3

3	1	2	−3
−1	2	1	−3
1	1	−1	1

- For each window of k words
  - Compute Convolution between
    1. concatenation of word vectors
    2. and filter

10

Convolution example (from C.Manning) (2/5)

1D convolution for text with padding

ø	0.0	0.0	0.0	0.0
tentative	0.2	0.1	−0.3	0.4
deal	0.5	0.2	−0.3	−0.1
reached	−0.1	−0.3	−0.2	0.4
to	0.3	−0.3	0.1	0.1
keep	0.2	−0.3	0.4	0.2
government	0.1	0.2	−0.1	−0.1
open	−0.4	−0.4	0.2	0.3
ø	0.0	0.0	0.0	0.0

ø,t,d	−0.6
t,d,r	−1.0
d,r,t	−0.5
r,t,k	−3.6
t,k,g	−0.2
k,g,o	0.3
g,o,ø	−0.5

Apply a **filter** (or **kernel**) of size 3

3	1	2	−3
−1	2	1	−3
1	1	−1	1

Padding

- compute word window representations
- for border positions

11

Convolution example (from C.Manning) (3/5)

3 channel 1D convolution with padding = 1

ø	0.0	0.0	0.0	0.0
tentative	0.2	0.1	−0.3	0.4
deal	0.5	0.2	−0.3	−0.1
reached	−0.1	−0.3	−0.2	0.4
to	0.3	−0.3	0.1	0.1
keep	0.2	−0.3	0.4	0.2
government	0.1	0.2	−0.1	−0.1
open	−0.4	−0.4	0.2	0.3
ø	0.0	0.0	0.0	0.0

ø,t,d	−0.6	0.2	1.4
t,d,r	−1.0	1.6	−1.0
d,r,t	−0.5	−0.1	0.8
r,t,k	−3.6	0.3	0.3
t,k,g	−0.2	0.1	1.2
k,g,o	0.3	0.6	0.9
g,o,ø	−0.5	−0.9	0.1

Apply 3 **filters** of size 3

3	1	2	−3
−1	2	1	−3
1	1	−1	1

1	0	0	1
1	0	−1	−1
0	1	0	1

1	−1	2	−1
1	0	−1	3
0	2	2	1

Could also use (zero) padding = 2

Also called “wide convolution”

- Several filters applied in parallel
- Compute vector representations for windows

Convolution example (from C.Manning) (4/5)

conv1d, padded with max pooling over time

ø	0.0	0.0	0.0	0.0
tentative	0.2	0.1	−0.3	0.4
deal	0.5	0.2	−0.3	−0.1
reached	−0.1	−0.3	−0.2	0.4
to	0.3	−0.3	0.1	0.1
keep	0.2	−0.3	0.4	0.2
government	0.1	0.2	−0.1	−0.1
open	−0.4	−0.4	0.2	0.3
ø	0.0	0.0	0.0	0.0

ø,t,d	−0.6	0.2	1.4
t,d,r	−1.0	1.6	−1.0
d,r,t	−0.5	−0.1	0.8
r,t,k	−3.6	0.3	0.3
t,k,g	−0.2	0.1	1.2
k,g,o	0.3	0.6	0.9
g,o,ø	−0.5	−0.9	0.1

max p	0.3	1.6	1.4

Apply 3 **filters** of size 3

3	1	2	−3
−1	2	1	−3
1	1	−1	1

1	0	0	1
1	0	−1	−1
0	1	0	1

1	−1	2	−1
1	0	−1	3
0	2	2	1

- Pooling
  1. Select n values from each dimension
     - (usually $n = 1$)
     - order in sequence irrelevant
  2. output fixed-size vector

Convolution example (from C.Manning) (5/5)

conv1d, padded with ave pooling over time

ø	0.0	0.0	0.0	0.0
tentative	0.2	0.1	−0.3	0.4
deal	0.5	0.2	−0.3	−0.1
reached	−0.1	−0.3	−0.2	0.4
to	0.3	−0.3	0.1	0.1
keep	0.2	−0.3	0.4	0.2
government	0.1	0.2	−0.1	−0.1
open	−0.4	−0.4	0.2	0.3
ø	0.0	0.0	0.0	0.0

ø,t,d	−0.6	0.2	1.4
t,d,r	−1.0	1.6	−1.0
d,r,t	−0.5	−0.1	0.8
r,t,k	−3.6	0.3	0.3
t,k,g	−0.2	0.1	1.2
k,g,o	0.3	0.6	0.9
g,o,ø	−0.5	−0.9	0.1

ave p	−0.87	0.26	0.53

Apply 3 **filters** of size 3

3	1	2	−3
−1	2	1	−3
1	1	−1	1

1	0	0	1
1	0	−1	−1
0	1	0	1

1	−1	2	−1
1	0	−1	3
0	2	2	1

Pooling

- different operations possible
  - max
  - min
  - avg?
  - sum??

14

2 Convolution example Pytorch (1)

```
import torch
import torch.nn as nn

batch_size, word_embed_size, sent_len = 2, 4, 5
data = torch.randn(batch_size, word_embed_size, sent_len)
conv1 = nn.Conv1d(in_channels=word_embed_size, out_channels=2,
                  kernel_size=3, padding=1)
mp = nn.AdaptiveMaxPool1d(output_size=1)

print(data.size())
print(data)
```

```
A module that was compiled using NumPy 1.x cannot be run in
NumPy 2.2.6 as it may crash. To support both 1.x and 2.x
versions of NumPy, modules must be compiled with NumPy 2.0.
Some module may need to rebuild instead e.g. with 'pybind11>=2.12'.

If you are a user of the module, the easiest solution will be to
downgrade to 'numpy<2' or try to upgrade the affected module.
We expect that some modules will need time to support NumPy 2.

Traceback (most recent call last):  File "<stdin>", line 1, in <module>
  File "<string>", line 17, in __PYTHON_EL_eval
  File "<string>", line 3, in <module>
```

2 Convolution example Pytorch (2)

```
h1 = conv1(data)
h2 = mp(h1)

print(h1.size())
print(h2.size())
print(h1)
print(h2)
```

```
torch.Size([2, 2, 5])
torch.Size([2, 2, 1])
tensor([[[-0.7823, -0.7308, -0.0213, -0.0638, -0.1242],
         [ 1.3488,  0.0778,  0.0357, -0.3898, -0.6341]],

        [[-0.5564,  0.1718,  0.1567,  0.6167, -0.1108],
         [-0.0120,  0.6213, -0.4980, -0.9997, -0.8356]]],
       grad_fn=<ConvolutionBackward0>)
tensor([[[-0.0213],
         [ 1.3488]],

        [[ 0.6167],
         [ 0.6213]]], grad_fn=<SqueezeBackward1>)
```

Stride (step size)

- no obligation to consider consecutive positions (stride=1)

Dilation

- stacking convolution on top of a convolution
- each time sequence length decreases

Attention mechanism

- why limit the size of the filter? (cf. lecture on Transformers)

2 Wrap-up : CNN-based sentence classifier

Summary

- a convolution of size m with f filters on word vectors of a sentence
- pipe to a pooling operation to get a fixed size vector v
- pipe result v to a MLP to classify into c classes
- parameter : MLP, convolution, word vectors
- learning parameters via MLE (also called cross-entropy)

Extension

Use several convolutions

1. different sizes (3,4,5...)
2. different *strides*
3. different dilations

Further readings

More details on CNN for text, with expeirmental setup described in [Kim14]

3 Bibliography

[Kim14] Yoon Kim. "Convolutional Neural Networks for Sentence Classification." In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751. DOI: 10.3115/v1/D14-1181. URL: https://aclanthology.org/D14-1181/.

4 Beyond Convolutions: a Brief History of Sequence Models

4 Recurrent Neural Networks (Historical Context)

Before Transformers: RNNs

- Recurrent Neural Networks process sequences step-by-step, maintaining a hidden state h_t:

$$h_t = \eta(Uh_{t-1} + Vx_t + b)$$

- Unlike CNNs, RNNs can (in principle) capture *unbounded* context.
- Used extensively 2013–2018 for language modeling, machine translation, sequence labeling.

Why not RNNs?

- Sequential computation: step t depends on $h_{t-1} \Longrightarrow$ hard to parallelize on GPUs.
- Vanishing gradients: the term U^{t-1} in the gradient grows or shrinks exponentially $\Longrightarrow$ distant words contribute almost nothing to learning.
- Variants (LSTM, GRU) help partially, but the fundamental bottleneck remains.

4 From RNNs to Transformers

The Core Problem

- CNNs: fixed, *local* context window.
- RNNs: unlimited context in theory, but gradients vanish *and* computation is serial.
- Both force us to compress the entire past into a single fixed-size vector.

The Key Idea: Attention

- What if, instead of compressing, we let each position directly attend to every other position?
- Attention weights are computed from the content of the sequence itself.
- Fully parallelizable: no recurrence, no sequential bottleneck.
- [⇒] Transformers (Vaswani et al., 2017) covered in the next lecture.

5
Cross-Entropy Learning

5 Cross-Entropy Learning

Cross-Entropy between discrete distributions

$$H(p,q) = -\sum_x p(x)\log\big(q(x)\big) = \mathbb{E}_{x\sim p(\cdot)}[-\log q(x)]$$

- positive (min when $p = q$)
- $\equiv$ *size* of approximating p with q (in number of bits when log_2)

As a loss function

Assume data distributed according to p, and our model implements q, we want to minimise $H(p,q)$. What is the best q?

- $\{(x_i, k_i)\}$ distributed as p : $p(k_i|x_i) = 1$ and $p(k'|x_i) = 0$ if $k' \neq k_i$

$$\begin{aligned} H(p,q) &= -\sum_{(x,k)} p(k|x)\log\big(q(k|x)\big) \\ &= -\sum_{(x_i,k_i)} p(k_i|x_i)\log\big(q(k_i|x_i)\big) \\ &= \sum_{(x_i,k_i)} -\log\big(q(k_i|x_i)\big) = \text{- log-likelihood!!} \end{aligned}$$

5 Cross-Entropy in Pytorch

```
batch_size = 3
nb_classes = 5
loss = nn.CrossEntropyLoss()
#random predictions // should be the result of our model (NN)
prediction = torch.randn(batch_size, nb_classes, requires_grad=True)
#random classes // should be the data example labels
true_classes = torch.empty(3, dtype=torch.long).random_(5)
#cross-entropy // log-likelihood of predictions
err = loss(prediction, true_classes)
#err.backward()
#...
```

`CrossEntropyLoss` computes:

1. softmax of predictions
2. entropy

5 More generally...

With Pytorch (and other libraries)

- separate computations for predictions (*forward*) et errors (*loss*)
- implement the prediction part
- use a predefined loss function

6
Practical GD: Learning Rate

6 Recall (1/2) [Learning via GD DM-INFO3]

(To simplfy things, assume that R approximates f a function of $\mathbb{R}^n \longrightarrow \mathbb{R}$)
We will solve our problem (find a local minimum $L(\theta)$) by an iterative method:

Gradient Descent Algorithm

- Initialise θ^0 randomly
- $t = 0$
- While $\|\nabla L(\theta^t))\|_2 \geq \epsilon$
  - Set step size $\alpha_t > 0$
  - $\theta^{t+1} = \theta^t - \alpha_t \nabla L(\theta_t)$
  - t = t+1

Simple, provided we can compute the gradient, but why does it work??

6 Recalls (2/2) [Learning via GD DM-INFO3]

Easy part, each iteration improves the solution (less errors on training examples). First-order Taylor expansion of L around θ^t:

$$\begin{aligned} L(\theta^t - \alpha_t \nabla L(\theta_t)) &= L(\theta^t) + \langle \nabla L(\theta_t), (\theta_t - \alpha_t \nabla L(\theta_t)) - \theta_t \rangle + o(|\alpha_t \nabla L(\theta_t)|) \\ &= L(\theta^t) - \alpha_t \|\nabla L(\theta_t)\|^2 + o(|\alpha_t \nabla L(\theta_t)|) \\ &\approx L(\theta^t) - \alpha_t \|\nabla L(\theta_t)\|^2 \text{ for } \alpha_t \text{ small enough} \\ &\leq L(\theta^t) \end{aligned}$$

The other part, (improvement is real, and we can bound the number of steps to reach the local minimum) needs more hypotheses

6 How to set α_t?

In some cases there exists α that leads to the optimal parameters

(Second-order Taylor expansion + hypotheses...)

- but not for us, we are stuck with stochastic gradient descent

In practice

1. constant step $\alpha_t = s \forall t$
   - simple but may fail to converge (or too slow)
2. decreasing step $\lim_\infty a_t = 0, \sum \alpha_t = \infty$ (eg $\alpha_t = \frac{1}{t}$)
   - converges *eventually* but may be slow
3. by minimisation $\alpha^* = \arg\min_\alpha L(\theta_k - \alpha \nabla L(\theta_k))$
   - gives the *best* step
   - a 2nd problem to solve ! (slow and/or difficult)
   - can be approximated in practice
4. by successive decrease steps
   - compromise between 1 and 2, usual method for NNn training.

6 Update α in pytorch (1)

Via scheduler subclases

```
scheduler = ... #object creation
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...) # depends on subclasses
>>>     scheduler.step() # at each epoch
```

torch.optim.lr_scheduler.LambdaLR

takes as imput a function

1. that takes as input the epoch evaluation value
2. that returns a multiplcator coeff the initial α

6 Update α in pytorch (2)

`torch.optim.lr_scheduler.MultiplicativeLR` and `torch.optim.lr_scheduler.StepLR`

- simplified versions of `LambdaLR`
- that take as input scalars (not function)

`torch.optim.lr_scheduler.ReduceLROnPlateau`

- take into account dev score to change α

7

Practical GD: Regularisation and Dropout

7 When is learning θ over ?

Answer 1

When $L = 0$!

Answer 2

When L is minimised!

Answer 3

... it depends ...

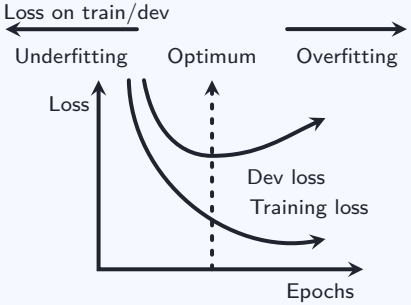
Final Answer

- We want parameters that will behave correctly with new data, data with which it will be used
- That is the role of dev and test sets to represent such data !

7 Problem of *Overfitting*

- With NNs it is easy to add many parameters
- At the limit, enough parameters to memorize the training data
- Problem: in this case, the NNs cannot cope with new data, it not generalising.

Typical illustrations of overfitting:



(taken from `https://tex.stackexchange.com`)

7 How to avoid overfitting?

Idea: Prevent the NN to minimise L

1. simplify you NN (meh...)
2. change the loss function to prevent memorisation
3. parturb the training process to make it robust to noise/small variations

7 Regularisation (1/2)

Change objective to prevent memorisation

- idea: prevent θ to completely fit data
- how: by constraining θ usually by limiting its norm
- several choices, usually norm "L2"

We want to mimise:

$$J(\theta) = L(\theta) + \frac{C}{2}||\theta||_2^2$$

where L is a loss (eg cross-entropy)

Easy to implement in Gradient Descent

the loss gradient becomes:

$$\nabla J(\theta) = \nabla L(\theta) + C\theta$$

7 Regularisation (2/2)

Intuition

- at each step of SGD, we substract C to all parameter values.
- less differences between θ_i (they're all small)
- only θ_i updated in L very frequently can be increased
- tend to *simplify* the model

In pytorch:

parameter `weight_decay` of the `optimizer`

7 Perturbation of the network (1/4)

Idea: make some noise!

For instance, with a Gaussian distribution:

```
import torch as t
class NoiseMLP(t.nn.Module):
    def __init__(self, in_dim, hidden_dim, out_dim):
        super(NoiseMLP,self).__init__()
        self.linear1 = t.nn.Linear(in_dim, hidden_dim)
        self.linear2 = t.nn.Linear(hidden_dim, out_dim)
        self.normal_dist = t.distributions.Normal(loc=t.tensor([0.]),
                                                  scale=t.tensor([1.0]))

    def forward(self, x):
        x = self.linear1(x)
        ##True by default, controlled by methods train() and eval()
        if self.training:
            print(x) # never print in forward in production
            noise = self.normal_dist.sample(x.size()).squeeze()
            print(noise) # never print in forward in production
            x = x + noise
            print(x) # never print in forward in production
        return self.linear2(t.tanh(x))
```

7 Perturbation of the network (2/4)

```
nmlp = NoiseMLP(10,5,1)
input = t.randn(2,10)
r = nmlp(input)
```

```
tensor([[-0.7477,  0.1289, -1.2526,  0.0030,  0.8901],
        [-0.4431,  0.7191, -1.2379,  0.2766, -1.1907]],
       grad_fn=<AddmmBackward0>)
tensor([[ 1.1135, -1.4081,  0.4912,  0.4083, -0.6487],
        [ 2.2599, -0.8356,  0.7692, -0.3998,  0.8962]])
tensor([[ 0.3658, -1.2792, -0.7614,  0.4112,  0.2414],
        [ 1.8168, -0.1165, -0.4687, -0.1232, -0.2946]], grad_fn=<AddBackward0>)
```

7 Perturbation of the network (3/4)

Idea: Block random neurons

- Replace intermediate by zeros randomly
- Called the dropout method
- ≈ sampling subsets of parameters for eahc examples

```
import torch
class DoMLP(torch.nn.Module):
    def __init__(self, in_dim, hidden_dim, out_dim):
        super(DoMLP,self).__init__()
        self.linear1 = torch.nn.Linear(in_dim, hidden_dim)
        self.linear2 = torch.nn.Linear(hidden_dim, out_dim)
        self.dropout = torch.nn.Dropout(p=0.5)

    def forward(self, x):
        x = self.linear1(x)
        print(x)
        x = self.dropout(x)
        print(x)
        return self.linear2(torch.tanh(x))
```

7 Perturbation of networks (4/4)

```
dmlp = DoMLP(10,5,1)
input = torch.randn(2,10)
r = dmlp(input)
```

```
tensor([[ 0.8663, -0.6211,  0.9426,  0.0513,  0.1932],
        [ 0.8466, -0.1701,  0.2814,  0.8139,  0.2694]],
       grad_fn=<AddmmBackward0>)
tensor([[ 0.0000, -0.0000,  1.8852,  0.0000,  0.0000],
        [ 0.0000, -0.3402,  0.0000,  0.0000,  0.5388]], grad_fn=<MulBackward0>)
```