Convolutional Networks for NLP and Complements on Training

Joseph Le Roux

18/08/26

Notes

Outline

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

Notes

1 Motivation

Notes

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 $\mathbb{R}^d \longrightarrow \mathbb{R}^c$
- 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)

Notes

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"

Notes

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

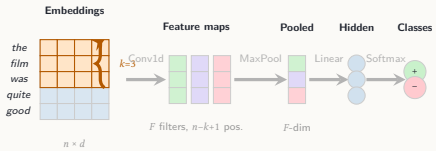
- sentence of n words represented by a vector in $\mathbb{R}^{nd}$
- problem: no unique MLP can transform a vector for a sentence into a vector of class scores, requires one MLP per sentence length.
- but still... concatenation is an important idea

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

Notes

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

Notes

2 Convolutional Networks

Notes

2 Convolutions : CNNs

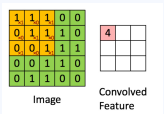
Problem with MLPs: unstructured representation

- 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



From Stanford UFLDL wiki

Notes

2 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

Notes

2 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

11

Padding

- compute word window representations
- for border positions

Notes

2 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

Notes

2 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

Notes

2 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??

Notes

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)
```

```
torch.Size([2, 4, 5])
tensor([[[ 2.0870,  0.9029, -2.0567,  0.6988, -2.1653],
         [ 0.6180, -0.9951, -1.1217, -1.4111, -0.7387],
         [ 0.1048,  1.3269,  0.7464, -0.3848, -1.3867],
         [-1.0279, -0.5564, -1.3459,  0.7254, -0.2374]],

        [[-0.1131, -1.6291, -1.4130, -0.0908,  0.1451],
         [-0.5581, -0.9953,  0.3597,  1.2926,  0.0382],
         [ 1.0622,  0.1046,  0.0037,  0.7399, -0.9590],
         [ 0.8385, -2.1493, -0.1113,  0.0313,  0.1759]]])
```


Notes

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([[[-0.3808, -0.4130,  0.4834, -0.1059,  0.5186],
         [-0.1286, -0.3039, -0.9157,  0.3831, -0.4825]],

        [[-0.1924,  0.3185,  1.0654,  0.1878, -0.0940],
         [ 0.5962, -0.6351, -0.2343, -0.0856,  0.2300]]],
       grad_fn=<ConvolutionBackward0>)
tensor([[[0.5186],
         [0.3831]],

        [[1.0654],
         [0.5962]]], grad_fn=<SqueezeBackward1>)
```


Notes

2 Convolutions : extensions

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)

Notes

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]

Notes

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/.

Notes

4 Beyond Convolutions: a Brief History of Sequence Models

Notes

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

Notes

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.

Notes

5 Cross-Entropy Learning

Notes

Cross-Entropy between discrete distributions

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

- positive (min when $p = q$)
- = *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 $k' \neq k_i$

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

Notes

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

Notes

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

Notes

6
Practical GD: Learning Rate

Notes

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

(To simplfy things, assume that R approximates f a function of $\mathbb{R}^n \to \mathbb{R}$)
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??

Notes

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

Notes

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.

Notes

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

Notes

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

Notes

7 Practical GD: Regularisation and Dropout

Notes

7 When is learning θ over ?

Answer 1

When $L = 0$!

Answer 2

When L is minimised!

Answer 3

... it depends ...

Final Answer

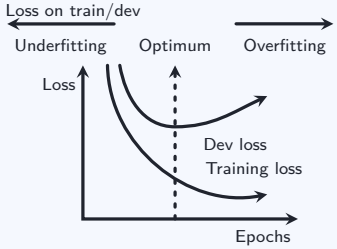
- We want parameters that will behave correctly with new data, data with which it will be used
- That is the role of dev and test sets to represent such data !

Notes

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`)

Notes

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

Notes

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:

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

Notes

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`

Notes

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))
```


Notes

7 Perturbation of the network (2/4)

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

```
tensor([[-0.4400,  0.0185,  0.3401, -0.3802,  0.4859],
        [-0.4613,  0.4717,  1.0660, -0.1380, -0.7464]],
       grad_fn=<AddmmBackward0>)
tensor([[-0.8726,  0.7081,  0.8039,  1.0777,  1.3873],
        [ 1.0585, -1.6333,  0.3512, -1.1195,  0.3914]])
tensor([[-1.3126,  0.7266,  1.1440,  0.6975,  1.8733],
        [ 0.5972, -1.1616,  1.4172, -1.2574, -0.3550]], grad_fn=<AddBackward0>)
```


Notes

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))
```


Notes

7 Perturbation of networks (4/4)

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

```
tensor([[ 0.8070, -0.7960, -0.5660,  0.6492,  0.4397],
        [ 0.2354,  0.4502, -0.4120,  0.1244,  0.6079]],
       grad_fn=<AddmmBackward0>)
tensor([[ 1.6139, -1.5921, -0.0000,  0.0000,  0.8794],
        [ 0.4708,  0.0000, -0.0000,  0.2488,  0.0000]], grad_fn=<MulBackward0>)
```


Notes