Tutoriel Deep Learning: applications signal

Thomas Pellegrini

Université de Toulouse; UPS; IRIT; Toulouse, France
CCT TSI 26 janvier 2017
Multiple Layers of simple units

Each units computes a weighted sum of its inputs

Weighted sum is passed through a non-linear function

The learning algorithm changes the weights

ReLU (x) = max(x, 0)
Gradients

- A practical Application of Chain Rule
  - Backprop for the state gradients:
    - $\frac{dC}{dX_{i-1}} = \frac{dC}{dX_i} \cdot \frac{dX_i}{dX_{i-1}}$
    - $\frac{dC}{dX_{i-1}} = \frac{dC}{dX_i} \cdot \frac{dF_i(X_{i-1}, W_i)}{dX_{i-1}}$
  - Backprop for the weight gradients:
    - $\frac{dC}{dW_i} = \frac{dC}{dX_i} \cdot \frac{dX_i}{dW_i}$
    - $\frac{dC}{dW_i} = \frac{dC}{dX_i} \cdot \frac{dF_i(X_{i-1}, W_i)}{dW_i}$

$F(X_{i-1}, W_i)$

- $F_n(X_{n-1}, W_n)$
- $F_1(X_0, W_1)$

$C(X, Y, \Theta)$

$X$ (input)

$Y$ (desired output)

[Y. LeCun]
Affine layer: forward

\[ Y = X \cdot W + b \]

```
def affine_forward(x, w, b):
    out = np.dot(x, w) + b
    cache = (x, w, b)
    return out, cache
```
Affine layer: backward

\[ dW = X^t \cdot dout \]
\[ db = \sum_{i=1}^{N} dout^i \]
\[ dx = dout \cdot W^t \]

```python
def affine_backward(dout, cache):
    x, w, b = cache
    dx = np.dot(dout, w.T)
    dw = np.dot(x.T, dout)
    db = np.sum(dout, axis=0)
    return dx, dw, db
```
Non-linearity layer: ReLu forward

\[ Y = \max(0, X) \]
\[ = X * 1_{\{X>0\}} \]
\[ = X * [X > 0] \]

```python
def relu_forward(x):
    out = np.maximum(np.zeros((x.shape)), x)
    cache = x
    return out, cache
```
Non-linearity layer: ReLu backward

\[ dx = [X > 0] \ast dout \]

```python
def relu_backward(dout, cache):
    x = cache
    dx = dout * ((x>0)*1)
    return dx
```
Dropout layer: forward

$r_j \sim \text{bernoulli}(p)$

\[ Y = R \ast X \]

```python
def dropout_forward(x, p, mode):
    if mode == 'train':
        mask = (np.random.rand(*x.shape) < p) * 1
        out = x * mask
    elif mode == 'test':
        out = x
    cache = (p, mode, mask)
    out = out.astype(x.dtype, copy=False)
    return out, cache
```
Dropout layer: backward

\[ dx(NxM) \rightarrow \text{dropout} \rightarrow R(NxM) \rightarrow dout (NxM) \]

\[ dx = R \ast dout \]

```python
def dropout_backward(dout, cache):
    p, mode, mask = cache
    if mode == 'train':
        dx = dout * mask
    elif mode == 'test':
        dx = dout
    return dx
```
Batch-normalization layer

Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...m\}$; Parameters to be learned: $\gamma, \beta$

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$  // mini-batch mean

$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$  // mini-batch variance

$\tilde{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$  // normalize

$y_i \leftarrow \gamma \tilde{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$  // scale and shift
Batch-normalization layer

\[
\begin{align*}
\chi & \rightarrow - & \sum_{k} \frac{1}{N} & \rightarrow \sqrt{x - \epsilon} & \rightarrow \frac{1}{\sqrt{x}} \\
\gamma & \rightarrow * & \frac{1}{\sqrt{N}} & \rightarrow * & + \\
\beta & \rightarrow + & \text{out} & \rightarrow \text{dout}
\end{align*}
\]
def batchnorm_forward(x, gamma, beta, bn_param):
    mode = bn_param['mode']
    eps = bn_param.get('eps', 1e-5)
    momentum = bn_param.get('momentum', 0.9)

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

    if mode == 'train':
        moy = np.mean(x, axis=0)
        var = np.var(x, axis=0)
        num = x - moy
        den = np.sqrt(var + eps)
        x_hat = num / den
        out = gamma * x_hat + beta
        running_mean = momentum * running_mean + (1. - momentum) * moy
        running_var = momentum * running_var + (1. - momentum) * var
        cache = (x, gamma, beta, eps, moy, var, num, den, x_hat)
    elif mode == 'test':
        x_hat = (x - running_mean)/np.sqrt(running_var + eps)
        out = gamma * x_hat + beta
        cache = (x, gamma, beta)
    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var
    return out, cache
Batch-normalization layer: backward with running mean

```python
def batchnorm_backward(dout, cache):
    x, gamma, beta, eps, moy, var, num, den, x_hat = cache
    dbeta = np.sum(dout, axis=0)
    dgamma = np.sum(dout * x_hat, axis=0)

    dxhat = gamma * dout
    dnum = dxhat / den
    dden = np.sum(-1.0 * num / (den**2) * dxhat, axis=0)

    dmu = np.sum(-1.0 * dnum, axis=0)
    dvareps = 1.0 / (2 * np.sqrt(var + eps)) * dden

    N, D = x.shape
    dx = 1.0 / N * dmu + 2.0 / N * (x - moy) * dvareps + dnum

    return dx, dgamma, dbeta```

From scores to probabilities

scores: \( f = F_n(X_{n-1}, W_n) \)

Probability associated to a given class \( k \):

\[
P(y = k|W, X) = \frac{\exp(f_k)}{\sum_{j=0}^{C-1} \exp(f_j)} = \text{softmax}(f, k)
\]

```python
def softmax(z):
    '''z: a vector or a matrix z of dim C x N'''
    z = z - np.max(z)  # to avoid overflow with exp
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=0)
```
Categorical cross-entropy loss

\[ \mathcal{L}(W) = -\frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(W|y^i, x^i) \]

\[ \mathcal{L}(W|y^i, x^i) = -\log(P(y^i|W, x^i)) \]

Only the probability of the correct class is used in \( \mathcal{L} \)
Categorical cross-entropy loss: gradient

\[ \nabla_{W_k} \mathcal{L}(W|y^i, x^i) = \frac{\partial \mathcal{L}(W|y^i, x^i)}{\partial W_k} \]

\[ = - \sum_{j=0}^{C-1} t^i_j \frac{\partial \log(z^i_j)}{\partial W_k} \]

with \( t^i_j = \mathbb{1}\{y^i=j\} \)

\[ = - \sum_{j=0}^{C-1} t^i_j \frac{1}{z^i_j} \frac{\partial z^i_j}{\partial W_k} \]

\[ = \ldots \]

\[ = -x^i(t^i_k - z^i_k) \]

\[ = \begin{cases} 
    x^i(z^i_k - 1) & \text{if } t^i_j = 1 \ (\text{i.e., } y^i = k) \\
    x^i z^i_k & \text{if } t^i_j = 0 \ (\text{i.e., } y^i \neq k) 
\end{cases} \]
def softmax_loss_vectorized(W, X, y, reg):
    """
    Softmax loss function, vectorized version.
    Inputs: same as softmax_loss_naive
    """
    # Initialize the loss and gradient to zero.
    loss = 0.0
    dW = np.zeros_like(W)
    D, N = X.shape
    C, _ = W.shape

    probs = softmax(W.dot(X)) # dim: C, N
    probs = probs.T # dim: N, C
    # compute loss only with probs of the training targets
    loss = np.sum(-np.log(probs[range(N), y]))
    loss /= N
    loss += 0.5 * reg * np.sum(W**2)

    dW = probs # dim: N, C
    dW[range(N), y] -= 1
    dW = np.dot(dW.T, X.T)
    dW /= N
    dW += reg * np.sum(W)

    return loss, dW
Our first modern network!

```python
def affine_BN_relu_dropout_forward(x, w, b, gamma,
    beta, bn_param, p, mode):

    network, fc_cache = affine_forward(x, w, b)
    network, bn_cache = batchnorm_forward(network, gamma,
        beta, bn_param)

    network, relu_cache = relu_forward(network)
    network, dp_cache = dropout_forward(network, p, mode)

    cache = (fc_cache, bn_cache, relu_cache, dp_cache)

    return network, cache

def affine_BN_relu_dropout_backward(...):
    ...
```
Our first modern network! Easier with a toolbox...

```python
from lasagne.layers import InputLayer,.DenseLayer, NonlinearityLayer, BatchNormLayer, DropoutLayer
from lasagne.nonlinearities import softmax

net = {}
net['input'] = InputLayer((None, 3, 32, 32))
net['aff'] = DenseLayer(net['input'],
                        num_units=1000, nonlinearity=None)
net['bn'] = BatchNormLayer(net['aff'])
net['relu'] = NonlinearityLayer(net['bn'])
net['dp'] = DropoutLayer(net['relu'])
net['prob'] = NonlinearityLayer(net['dp'], softmax)
```
Questions

- Which features are typically used as input?
- How to choose and design a model architecture?
- How to get a sense of what a model did learn?
- What is salient in the input that makes a model take a decision?

Examples in speech and singing birds
What features are typically used as input?

In audio applications: (log Mel) filter-bank coefficients most used!

Others:

- Raw signal
- FFT coefficients (module)
- MFCCs usually outperformed by F-BANK coefficients
Phone recognition: DNN

DNN Architecture

Input layer
11 frames of 24-dimensional log Mel filter bank coefficients + deltas

5 sigmoid hidden layers
256 nodes each; fully connected feed-forward

Softmax output layer
41 nodes for 40 phonemes and silence; context independent

[Nagamine et al., IS 2015; Slide by T. Nagamine]
Phone recognition: CNN

[Abdel-Hamid et al., TASLP 2014]
Convolution maps

[Convolution maps with /i/, /e/, /E/, /o/, /O/, /a/, /u/ in F1 (Hz) and F2 (Hz) axes as per Pellegrini & Mouysset, IS 2016]
Convolution maps

K-means

Spectral clustering

F-measure vs. layers for different techniques.

[1] Pellegrini & Mouysset, IS 2016
Phone recognition: CNN with raw speech

Conv 1
$kW = 30$
$dW = 10$

MP 1
$kW = 2$
$dW = 2$

Conv 2
$kW = 5$
$dW = 1$

MP 2
$kW = 2$
$dW = 2$

Conv 3
$kW = 5$
$dW = 1$

MP 3
$kW = 2$
$dW = 2$

ANN
$p(i|x)$

[Magimai-Doss et al., IS 2013 ; Slide by M. Magimai-Doss]
Phone recognition: CNN with raw speech

The first convolution layer can be seen as a bank of matching filters.

- Standard Mel filterbanks: 20-30 ms window
- Filters learned by the CNN: 2-4 ms window, 0.6 ms
- Found empirically, based on performance on the validation set.
- These filters respond to different frequency bands:

![Graphs showing frequency response of timers for filters 43, 73, and 67.](image)

[Magimai-Doss et al., IS 2013 ; Slide by M. Magimai-Doss]
Phone recognition: CNN with raw speech

- Filter cumulative response: sum of all filters magnitude spectrum.
Phone recognition: CNN with raw speech

- Filter cumulative response: sum of all filters magnitude spectrum.

[Magimai-Doss et al., IS 2013 ; Slide by M. Magimai-Doss]
Handling time series

- Frame with context: decision at frame-level
- Pre-segmented sequences: TDNN, RNN, LSTM
- Sequences with no previous segmentation: Connectionist Temporal Classification loss [Graves, ICML 2006]
Recent convNets architectures

- Standard convNets

\[ x_i = F_i(x_{i-1}) \]

Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

[He et al, CVPR 2016]
Recent convNets architectures

- Standard convNets [LeCun, 1995]
  \[ x_i = F_i(x_{i-1}) \]

- Residual convNets [He et al, CVPR 2016]
  \[ x_i = F_i(x_{i-1}) + x_{i-1} \]

- Densely connected convNets [Huang et al, 2016]
  \[ x_i = F_i([x_0, x_1, \ldots, x_{i-1}]) \]
DenseNets: dense blocks
Bird Audio Detection challenge 2017
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Train</th>
<th>Valid</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freefield1010</td>
<td>6,152</td>
<td>384</td>
<td>1,154</td>
</tr>
<tr>
<td>Warblr</td>
<td>6,800</td>
<td>500</td>
<td>700</td>
</tr>
<tr>
<td>Merged</td>
<td>14,806</td>
<td>884</td>
<td>0</td>
</tr>
<tr>
<td>Tchernobyl</td>
<td>_</td>
<td>_</td>
<td>8,620</td>
</tr>
</tbody>
</table>
Proposed solution: denseNets

- 74 layers
- 328K parameters
- Tchernobyl ROC (AUC) score: 88.79%

- Code densenet + saliency:
  [https://github.com/topel/](https://github.com/topel/)

- Audio + saliency map examples:
  [https://goo.gl/chxOPD](https://goo.gl/chxOPD)
How to get a sense of what a model did learn?

- Analysis of the weights (plotting), activation maps
- Saliency maps: which input elements (e.g., which pixels in case of an input image) need to be changed the least to affect the prediction the most?
Deconvolution methods

a) Input image $f^0$ → Forward pass $f^1, \ldots, f^{L-1}$ → Feature map $f^L$
Reconstructed image $R^0$ ← Backward pass $R^1, \ldots, R^{L-1}$ ← $R^L$

b) Forward pass
Backward pass: backpropagation
Backward pass: "deconvnet"
Backward pass: guided backpropagation

C) activation: $f_i^{l+1} = \text{relu}(f_i^l) = \max(f_i^l, 0)$
backpropagation: $R_i^l = (f_i^l > 0) \cdot R_{i+1}^{l+1}$, where $R_{i+1}^{l+1} = \frac{\partial f_{out}}{\partial f_{i+1}}$
backward "deconvnet": $R_i^l = (R_{i+1}^{l+1} > 0) \cdot R_i^{l+1}$
guided backpropagation: $R_i^l = (f_i^l > 0) \cdot (R_{i+1}^{l+1} > 0) \cdot R_i^{l+1}$

[Springenberg et al, ICLR 2015]
0070e5b1-110e-41f2-a9a5, P(bird): 0.966
<table>
<thead>
<tr>
<th>Reference</th>
<th>Conference/Year</th>
</tr>
</thead>
</table>