Ceng 783 – Deep Learning

Week 5 – Convolutional Neural Networks (continued)

Fall 2017

Emre Akbas
Misc

• Project proposal feedback
• Hw #2 is due on Monday.
So far:

Neural network

Input $x$

Estimated label $\hat{y}$

Matrix multiplication (FC), convolution

Linear and non-linear operations are stacked hierarchically. ReLU, sigmoid, pooling, etc.

Different connectivity types (fully, convolutional, local)
So far:

Estimated label \( \hat{y} \)

Input \( x \)

Neural network

Feedforward evaluation during testing/predicting
So far: training

\[
\text{Loss}(\hat{y}, y) \quad \text{Cross-entropy} \quad \text{Hinge loss}
\]

\[
\hat{y} \quad \text{Softmax}
\]

\[
x \quad \text{Input}
\]

\[
y \quad \text{True label}
\]

Gradient of loss w.r.t. parameters are computed using backpropagation.

Then, use a stochastic gradient descent method to minimize loss.
Today

Miscellaneous topics about ConvNets

- Loss/cost functions
  - Multiclass hinge
  - Where does cross-entropy come from?
- Implementing backpropagation
- Stochastic gradient descent variants
- How do I initialize my CNN?
- Batch normalization
Loss/cost functions
Multiclass hinge loss

[Crämer & Singer (2001)]

Training example \((x, y)\); neural network \(f()\)

\[ f : x \rightarrow C \text{ dimensional vector} \]

\[ \text{Hinge}(f(x), y) = \max \left( 0, 1 - f_y(x) + \max_{c \neq y} f_c(x) \right) \]
Cross-entropy

\[ L(\theta) = -\sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log q_{ic} \]

\( y_i \) is a C-dimensional one-hot vector
\( q_i \) is the softmax of \( f(x) \)

Q1: What does softmax do?

- Normalize the raw scores output by the neural network
- Highlight the max score

\[ q_{ic} = \frac{e^{f_c(x_i)}}{\sum_k e^{f_k(x_i)}} \]
Cross-entropy

\[ L(\theta) = - \sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log q_{ic} \]

\( y_i \) is a C-dimensional one-hot vector
\( q_i \) is the softmax of f(x)

Q2: Where does cross-entropy come from?
Cross-entropy

Definition:

\[ H(p, q) = - \sum_x p(x) \log q(x) \]

Minimized when \( p = q \)
Cross-entropy

Comes from maximum-likelihood estimation

[Derivation on board]
Cross-entropy loss as maximum likelihood (ML) estimation

Given a dataset $S = \{ \text{example}_i \}_{i=1}^N$ and a model,

$$P(S | \text{model}) = ? \quad \text{or} \quad P(S; \Theta) = ?$$

model's parameters

$$ML: \quad P(S; \Theta) = \frac{N}{\prod_{i=1}^{N} p(\text{example}_i; \Theta)}$$

i.i.d. assumption

Let's derive cross-entropy: We have a supervised dataset:

$S = \{ X, Y \}$ where $X$ matrix contains $N$ examples ($x_i$'s) and $Y$ contains the corresponding labels.

we want $\max_{\Theta} P(Y | X; \Theta)$

Use $ML \Rightarrow P(Y | X; \Theta) = \prod_{i=1}^{N} p(y_i | x_i; \Theta)$

$y_i$ is a one-hot vector and the model output is a $C$-dimensional $C$-dim vector containing estimated class probabilities.
\[
p(y_i | x_i) = p(y_{i1} | x_i)^{y_{i1}} p(y_{i2} | x_i)^{y_{i2}} \ldots p(y_{ic} | x_i)^{y_{ic}}
\]
\[
= \prod_{c=1}^{C} p(y_{ic} | x_i)^{y_{ic}}
\]

So ML is
\[
\max_{\Theta} \prod_{i=1}^{N} \prod_{c=1}^{C} p(y_{ic} | x_i)^{y_{ic}}
\]

Take log:
\[
\sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log p(y_{ic} | x_i)
\]

& multiply with -1 to make it a minimization problem:
\[
-\sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log p(y_{ic} | x_i)
\]

\[\text{can be implemented using softmax normalization}\]

\[
p(y_{ic} | x_i) = \frac{e^{f_c(x_i)}}{\sum_k e^{f_k(x_i)}}
\]

where \(f_c(x)\) is model's score for \(x\) belonging to class \(c\) output.
On the implementation of backpropagation

The “modular” approach
A neural network is nothing but a composition of several linear and non-linear functions:

\[ y = f_k (f_{k-1} (\ldots f_1 (x; \theta_1); \theta_{k-1}); \theta_k) \]

Given a specific architecture, i.e. composition, one can easily write the gradient w.r.t. parameters.

But a modular approach is desirable so that we don't have to derive the gradient again and again.

We can “compose” new architectures by simply connecting computing blocks.
A computing block:

Input $x$ → Function $f()$ → Output $o$

Params $w$

Forward pass: $o = f(x; w)$

Derivative of output w.r.t. input:

$$\frac{\partial o}{\partial x} = \frac{\partial f(x; w)}{\partial x}$$

Derivative of output w.r.t. parameters:

$$\frac{\partial o}{\partial w} = \frac{\partial f(x; w)}{\partial w}$$
A computing block:

\[
\begin{array}{c}
\text{Input } x \quad \rightarrow \quad \text{Function } f() \\
\quad \rightarrow \quad \text{Params } w \\
\quad \rightarrow \quad \text{Output } o
\end{array}
\]

Forward pass: \( o = f(x; w) \)

Derivative of output w.r.t. input:
\[
\frac{\partial o}{\partial x} = \frac{\partial f(x; w)}{\partial x}
\]

Derivative of output w.r.t. parameters:
\[
\frac{\partial o}{\partial w} = \frac{\partial f(x; w)}{\partial w}
\]

Typically, \( X, o \) and \( w \) are vectors or matrices. Care has to be taken in computing the derivatives.
**Exercise:** try to work out all the details for a fully connected layer with $D$ input nodes and $K$ output nodes, receiving $N$ examples.

Forward pass: $o = f(x; w)$

Derivative of output w.r.t. input:
$$\frac{\partial o}{\partial x} = \frac{\partial f(x; w)}{\partial x}$$

Derivative of output w.r.t. parameters:
$$\frac{\partial o}{\partial w} = \frac{\partial f(x; w)}{\partial w}$$
$X^*W$ where $W$ is D-by-K

$N$-by-$D$ \rightarrow X^*W$ where $W$ is D-by-K \rightarrow $N$-by-$K$

Derivative of output w.r.t. input: $\frac{\partial o}{\partial X} = W$

Derivative of output w.r.t. parameters: $\frac{\partial o}{\partial W} = X$
**Exercise:** do the same for a ReLU layer receiving N-by-K input

\[ \text{Max}(0, x) \]

X: N-by-K → Max(0, x) → o: N-by-K

Derivative of output w.r.t. input: ?

Derivative of output w.r.t. parameters: ?
Exercise: do the same for a ReLU layer receiving N-by-K input

\[ \text{Max}(0, x) \]

Derivative of output w.r.t. input:

\[ \frac{\partial o}{\partial x_{ij}} = \begin{cases} 1 & \text{if } x_{ij} > 0 \\ 0 & \text{otherwise} \end{cases} \]

Derivative of output w.r.t. parameters: No parameters, nothing to learn
Multiple blocks

\[ f_1(x; w_1) \rightarrow o_1 \rightarrow f_2(o_1; w_2) \rightarrow o_2 \]

To update \( w_2 \)
\[ \frac{\partial o_2}{\partial w_2} \]

To update \( w_1 \)
\[ \frac{\partial o_2}{\partial w_1} = \frac{\partial o_2}{\partial o_1} \frac{\partial o_1}{\partial w_1} \]

Each block has its own:
- Derivative w.r.t. input
- Derivative w.r.t. parameters.

When you are back-propagating, be careful which one to use.
Multiple blocks

\[
\frac{\partial o_3}{\partial w_1} = \frac{\partial o_3}{\partial o_2} \frac{\partial o_2}{\partial o_1} \frac{\partial o_1}{\partial w_1}
\]
Multiple blocks

\[ f_1(x; w_1) \rightarrow o_1 \rightarrow f_2(o_1; w_2) \rightarrow o_2 \rightarrow f_3(o_2; w_3) \rightarrow o_3 \]

\[ \frac{\partial o_3}{\partial w_1} = \frac{\partial o_3}{\partial o_2} \frac{\partial o_2}{\partial o_1} \frac{\partial o_1}{\partial w_1} \]

Last step: multiply with derivative w.r.t. parameters

Chain the "derivatives w.r.t. to input"
Exercise: work out all the details in a network where

- The input $X$ is $N$-by-$D$ ($N$ examples)
- The label vector is $N$-by-$K$ (in one-hot representation)
- There is a fully-connected layer and a softmax layer.
- Use cross-entropy as the loss.
Convolutional layer

- Can be implemented using matrix multiplication (instead of convolution)
  - Pros: we can make use of highly efficient linear algebra packages (e.g. BLAS, OpenBLAS)
  - Cons: same filter is repeated many times in the matrix → a lot of memory required

[example on board]
Convolutional layer as matrix multiplication

Example: suppose the input is 200x200 color images, hence 200x200x3

Conv layer: 75 filters of size 10x10x3. Stride 2

Take every block of 10x10x3 on the input image and turn it into a 300x1 vector.

Collect all such blocks and stack them in a matrix.
Convolutional layer as matrix multiplication

\[(200-10)/2 = 80\] locations in total.
So, 80x80 300-dim vectors. → \(X = 300 \times 6400\)

Conv filters: 75 filters of size 10x10x3: vectorize each filter and stack them row-by-row → \(W = 75 \times 300\)

\(W \times X \rightarrow 75 \times 6400\). Reshape this back to size (?)
Convolutional layer as matrix multiplication

\[(200-10)/2 = 80\] locations in total.
So, 80x80 300-dim vectors. → \(X = 300 \times 6400\)

Conv filters: 75 filters of size 10x10x3: vectorize each filter and stack them row-by-row → \(W = 75 \times 300\)

\(W \times X \rightarrow 75 \times 6400\). Reshape this back to size 80x80x75
Stochastic gradient descent variants
**Stochastic Gradient Descent (SGD)**

**Algorithm 8.1** Stochastic gradient descent (SGD) update at training iteration $k$

**Require:** Learning rate $\epsilon_k$.

**Require:** Initial parameter $\theta$

```
while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute gradient estimate: $\hat{g} \leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
    Apply update: $\theta \leftarrow \theta - \epsilon \hat{g}$
end while
```

[From Goodfellow et al. (2016)]
Setting the learning rate \((\alpha)\)

- **Constant learning rate:**
  - When it is low enough, guaranteed to make zero or positive progress.

- **Decaying learning rate:**
  - The rate is decreased as a function of the iteration number.
  - *Decaying is desired* because the gradient estimate is noisy due to the random sampling of \(m\) training examples. This noise does not vanish even when we arrive at a minimum.
  - By decreasing the rate, this source of noise is progressively eliminated.
Setting the learning rate ($\alpha$)

- Decaying learning rate:
  - Step decay
    - Reduce the rate by some factor every few epochs.
    - E.g. Halve the rate every 5 epochs → Numbers depend on the problem and dataset.
  - Exponential decay
    $$\alpha = \alpha_0 e^{-kt}$$
  - $1/t$ decay
    $$\alpha = \frac{\alpha_0}{1 + kt}$$

A complete pass through the dataset

[From http://cs231n.github.io/neural-networks-3/]

$\alpha_0, k$: hyperparameters

$\alpha$: iteration number
Setting the learning rate \((\alpha)\)

- Practical tips:
  - Step decay is slightly more preferable b/c its decrease schedule is more interpretable.
  - If you can afford, do slower decay and train for a longer time.

[From http://cs231n.github.io/neural-networks-3/]
Setting the learning rate \((\alpha)\)

- There are also second order methods:

\[
w^{t+1} \leftarrow w^t - \alpha \nabla_w f(x; w)
\]

where alpha is a function of the Hessian of \(f()\).

Newton method, BFGS, L-BFGS, Conjugate gradient

Not commonly used in deep learning.
Setting the learning rate ($\alpha$)

Practical tips

Adjust the hyper-parameters of your decaying schedule to get a “good” curve

[From http://cs231n.github.io/neural-networks-3/]
Setting the learning rate ($\alpha$)

Practical tips

Adjust the hyper-parameters of your decaying schedule to get a “good” curve.

[From http://cs231n.github.io/neural-networks-3/]
Setting the learning rate \((\alpha)\)

- A real example: training on CIFAR-10 dataset.
- Looks reasonable (maybe a little bit low alpha)
- Batch size might be a little too small (cost is very noisy).

[From http://cs231n.github.io/neural-networks-3/]

![Graph showing loss over epochs for the CIFAR-10 dataset training](attachment:loss_chart.png)
Setting the learning rate turned out to be really difficult.

So, researcher have develop **adaptive learning rate** methods
Adaptive learning rate methods

Momentum:

\[ v = mv - \alpha \nabla_w f(w) \]

Update: \[ w = w + v \]

Typical value for momentum is \( m=0.9 \)
Nesterov momentum

$$w^{\text{ahead}} = w + mv$$

$$v = mv - \alpha \nabla_{w^{\text{ahead}}} f(w^{\text{ahead}})$$

Update: $$w = w + v$$

Works slightly better than plain momentum. Getting more popular.

[From http://cs231n.github.io/neural-networks-3/]
Per parameter adaptive learning rate methods

• So far, learning rate was global (equally applied to all parameters in the model)

• Methods have been proposed to adapt learning rates per parameter.
  
  – Motivation: the cost function is highly sensitive in some directions and insensitive in others, in the parameter space
  
  – So, it might make sense to use different learning rates per parameter.
Per parameter adaptive learning rate methods

- E.g. “delta-bar-delta” algorithm [Jacobs (1988)]:

  “A heuristic method.

  If the partial w.r.t. to a given model parameter, remains the same sign, then the learning rate should increase.

  If it changes sign, then the learning rate should decrease. “

  [Goodfellow et al. (2016)]
Per parameter adaptive learning rate methods

- AdaGrad
- RMSprop
- Adam
Adagrad  [Duchi et al. (2011)]

```python
# Assume the gradient dx and parameter vector x
cache += dx**2
x += -learning_rate * dx / (np.sqrt(cache) + eps)
```

[From http://cs231n.github.io/neural-networks-3/]

Cache keeps track of per parameter sum of squared gradients.

Weights with high gradients $\rightarrow$ decrease learning rate
weights that receive small or infrequent updates $\rightarrow$ increase
learning rates.

AdaGrad performs well for some but not all deep learning models.
RMSprop
[Slide 29, Lecture 6, Hinton's Coursera class]

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += -learning_rate * dx / (np.sqrt(cache) + eps)
```

[From http://cs231n.github.io/neural-networks-3/]

Modifies Adagrad to use a moving average of squared gradients (instead of the complete sum over history).

Typically, \( \text{decay\_rate} = 0.9 \)

Performs very well. Definitely worth trying.
Adam

[Kingma and Ba (2014)]

\[
\begin{align*}
m &= \text{beta1} \cdot m + (1 - \text{beta1}) \cdot dx \\
v &= \text{beta2} \cdot v + (1 - \text{beta2}) \cdot (dx \cdot dx) \\
x &= x - \text{learning\_rate} \cdot \frac{m}{\text{np.sqrt(v) + eps}}
\end{align*}
\]

[From http://cs231n.github.io/neural-networks-3/]

Uses a “smoothed” gradient \(m\).

Recommended values for hyper-parameters: \(\text{eps}=1e^{-8}\), \(\text{beta1}=0.9\), \(\text{beta2}=0.999\)

In practice, Adam is currently recommended as the default method to use.

[From http://cs231n.github.io/neural-networks-3/]
Finally, when training, keep in mind the following:

Regularize more:

- Increase regularization's contribution to the cost function,
- Do more data augmentation.

[From http://cs231n.github.io/neural-networks-3/]
Very small gap between these two curves might also mean that you are underfitting, i.e. model capacity is too low → increase model capacity.

[From http://cs231n.github.io/neural-networks-3/]
References