Reminders

- Project proposal due on Oct 18.
- Hw2 will be out next week on Tuesday.
Recap (supervised learning)

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

\[ y \quad \text{Ground-truth label} \]

\[ \hat{y} = f(x; \theta) \quad \text{Model} \]

\[ L(\hat{y}, y) \quad \text{Loss} \]

\[ \arg \min_{\theta} R(\theta) + \lambda \sum_{(x, y)} L(f(x; \theta), y) \quad \text{Training. Minimize using 'gradient descent.'} \]
Linear SVM

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

\[ y \]  
\[ \text{Ground-truth label} \]

\[ \hat{y} = f(x; \theta) \]  
\[ \text{Linear model} \]

\[ L(\hat{y}, y) \]  
\[ \text{Hinge} \]

\[ \arg \min_{\theta} R(\theta) + \lambda \sum_{(x, y)} L(f(x; \theta), y) \]  
\[ \text{Training.} \]

\[ \text{L2 regularizer} \]

Gradient descent or custom methods
ANN with hidden layer(s)

\[ x \quad \text{Input} \]
\[ y \quad \text{Ground-truth label} \]
\[ \hat{y} = f(x; \theta) \quad \text{Non-linear, compositional model} \]
\[ L(\hat{y}, y) \quad \text{Cross-entropy (with softmax normalization)} \]

\[ \arg \min_{\theta} R(\theta) + \lambda \sum_{(x, y)} L(f(x; \theta), y) \quad \text{Training.} \]

Minimize using 'gradient descent.'

L1 or L2 or dropout regularization
Last time

- We implemented a hinge loss classifier.
- Now, let's add regularization to it and make it a proper Linear SVM. [go to Colab Notebook Week 3]
Today

- Biological neuron
- Artificial neuron
- Rosenblat's perceptron
- Multilayer Artificial Neural Network (ANN)
- Backpropagation
- Commonly used activation functions
- Stochastic Gradient Descent and Momentum
- A basic ANN example
Biological Neuron
Biological Neuron

“The job of a nerve cell is to take in information from the cells that feed into it, to sum up, or integrate, that information, and to deliver the integrated information to other cells. The information is usually conveyed in the form of brief events called nerve impulses.”

[Hubel (1995), Ch. 2]
Biological Neuron

Dendrites receive impulses from other cells.

Signal travels through the axon and is delivered to other cells via axon terminals through inter-cell regions called synapses.

[Figure from Hubel (1995)]
Biological neuron models

• A.k.a. spiking neuron models.
• A spike is a unitary event → all spikes have the same magnitude and duration.
• Spike = action potential
• Information is coded in the “rate” of the spikes.
  - Stimulus intensity increases → spike rate also increases [Adrian and Zotterman (1926); and then on]
Kuffler's work on cat retina, 1950s
Biological neuron models

• Many models
  – Integrate-and-fire model
  – Hodgkin-Huxley model
  – ...

“The physicochemical mechanisms of nerve and synaptic transmission are well understood. It should be obvious, however, that this kind of knowledge by itself cannot lead to an understanding of the brain, just as knowledge about resistors, condensers, and transistors alone will not make us understand a radio or TV.”

[Hubel (1995), Ch. 2]
Biological neuron population models

Spike train of an individual neuron also depends on its nearby neighbors → Better prediction accuracy than single neuron models.

[Pillow et al. (2008)]
Biological neural networks

[Hubel (1995), Ch. 2]
Figure 1.11: Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

[From Goodfellow et al. (2016)]
1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart et al., 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio et al., 1991)
7. Mean field sigmoid belief network (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998b)
10. Deep belief network (Hinton et al., 2006)
11. GPU-accelerated convolutional network (Chellapilla et al., 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina et al., 2009)
14. Unsupervised convolutional network (Jarrett et al., 2009)
15. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le et al., 2012)
18. Multi-GPU convolutional network (Krizhevsky et al., 2012)
19. COTS HPC unsupervised convolutional network (Coates et al., 2013)
20. GoogLeNet (Szegedy et al., 2014a)
Figure 1.10: Initially, the number of connections between neurons in artificial neural networks was limited by hardware capabilities. Today, the number of connections between neurons is mostly a design consideration. Some artificial neural networks have nearly as many connections per neuron as a cat, and it is quite common for other neural networks to have as many connections per neuron as smaller mammals like mice. Even the human brain does not have an exorbitant amount of connections per neuron. Biological neural network sizes from Wikipedia (2015).
1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla et al., 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett et al., 2009)
6. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
7. Distributed autoencoder (Le et al., 2012)
8. Multi-GPU convolutional network (Krizhevsky et al., 2012)
9. COTS HPC unsupervised convolutional network (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)
Artificial Neuron Models
Artificial Neuron

\[ o = \varphi \left( \sum_{i=1}^{D} w_i x_i; \theta \right) \text{ where } x \in \mathbb{R}^D, o \in \mathbb{R} \]
Artificial Neuron

\[ o = \varphi \left( \sum_{i=1}^{D} w_i x_i ; \theta \right) \text{ where } x \in \mathbb{R}^D, o \in \mathbb{R} \]
History of the artificial neuron

- 1943 – McCulloch-Pits Neuron
- 1958 – Perceptron
- 1960 – ADALINE
- 1986 – Backpropagation
McCulloch-Pits Neuron

- 1943
- Also called the Linear Threshold Logic Unit
- Binary input and output
- Can compute simple Boolean functions such as AND, OR, NOT
- No training!

Figure from http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html
Perceptron

- 1957 – Frank Rosenblat
- Real valued input
- For two-class classification

\[
\hat{y}_i = \begin{cases} 
1 & \text{if } w^T x_i > 0 \\ 
0 & \text{otherwise} 
\end{cases}
\]

- Proposed a learning algorithm: for each example, do

\[
w^{\text{next}} = w^{\text{curr}} + \eta (y_i - \hat{y}_i) x_i
\]
Why does perceptron work?

\[ w^{next} = w^{curr} + \eta(y_i - \hat{y}_i)x_i \]

If \( y_i = \hat{y}_i \), then \( x_i \) is correctly classified. Since \( y_i - \hat{y}_i = 0 \), \( w \) won't change.

For incorrectly classified \( x_i \):

Both \( y_i \) and \( \hat{y}_i \) are 0 or 1.

So, for \( y_i = 1 \), \( \eta(y_i - \hat{y}_i) > 0 \) and for \( y_i = 0 \), \( \eta(y_i - \hat{y}_i) < 0 \)

Multiply both sides of the learning rule by \( x_i^T \)

\[ x_i^T w^{next} = x_i^T w^{curr} + \eta(y_i - \hat{y}_i)x_i^T x_i \]

Since \( x_i^T x_i > 0 \), when \( y_i = 1 \) we have \( x_i^T w^{next} > x_i^T w^{curr} \)

when \( y_i = 0 \) we have \( x_i^T w^{next} < x_i^T w^{curr} \)
ADALINE

- 1960 – Widrow and Hoff
- Very similar to perceptron.
- Improved learning rule:

\[ w^{\text{next}} = w^{\text{curr}} + \eta(y_i - w^{\text{curr}} x_i)x_i \text{ where } y_i \in \{-1, +1\} \]

In perceptron, all errors were equal but in ADALINE, error is proportional to \((y_i - w^T x_i)\).

In fact, ADALINE does linear regression to the labels.
Backpropagation

• Rumelhart, Hinton and Williams (1986) is usually cited but in fact, backpropagation is nothing but an application of the chain rule.

• Backpropagation was invented again and again in different contexts (see this page).

• To compute the gradient of the error in the output layer, one has to compute the gradient iteratively layer by layer from the output layer to the input layer. (Hence, the name backpropagation).

• We'll get back to backprop details after introducing MLPs.
Multilayer Perceptron (MLP)
(Multilayer Artificial Neural Networks)
MLP

Fully connected, feedforward network

Input is a 4-dim vector.
There are 5 hidden nodes each with a 4-dim weight vector.
MLP

• Remember Cybenko's theorem?
• MLPs are universal function approximators.
• The most popular classification algorithm in 1980s (until SVMs were introduced)
MLP

Input $x \in \mathbb{R}^4$

$g(x; w_5) = \varphi(w_5^T x)$

Activation function

$f(x; \theta) = \sum_{i=1}^{5} \theta_i \varphi(w_5^T x)$
Question

• Simple models like the Perceptron (or ADALINE) cannot solve the XOR problem. (why?)

MLPs can solve it. Why?

Is it because of the hidden layers or the activation functions used?
Backpropagation

[Derivation on board]
0 = \varphi(\text{net}_0) = \varphi \left( \sum_{k=1}^{n} w_{0k} q_k \right)

\frac{\partial E}{\partial w_{0i}} = ?

\frac{\partial E}{\partial E_0} = \frac{\partial E_0}{\partial \text{net}_0} \frac{\partial \text{net}_0}{\partial w_{0i}}

depends on the activation function, e.g., sigmoid for

\sigma(1-\sigma)
Backpropagation

• Backprop gives you the gradient

• Then, apply a gradient descent method to minimize the cost function
  – We'll get back to this later
Backprop in deep networks

Should we be concerned about getting stuck at local minima?

“We identify a class of over-parameterized deep neural networks with standard activation functions and cross-entropy loss which provably have no bad local valley, in the sense that from any point in parameter space there exists a continuous path on which the cross-entropy loss is non-increasing and gets arbitrarily close to zero. This implies that these networks have no sub-optimal strict local minima.”

Nguyen et al., ICLR 2018 (Also see Nguyen et al. ICML 2017)
Backprop in deep networks

Should we be concerned about getting stuck at local minima?

“While the optimization problem behind deep neural networks is highly non-convex, it is frequently observed in practice that training deep networks seems possible without getting stuck in suboptimal points. It has been argued that this is the case as all local minima are close to being globally optimal. We show that this is (almost) true, in fact almost all local minima are globally optimal, for a fully connected network with squared loss and analytic activation function given that the number of hidden units of one layer of the network is larger than the number of training points and the network structure from this layer on is pyramidal.”

Nguyen et al. ICML 2017. (Also see Nguyen et al. ICML 2018)
• We have seen different loss functions before:
  – 0-1 loss, Hinge-loss, L_2 loss, cross-entropy loss, logistic loss

• Now let’s look at activation functions
  – Sigmoid, tanh, ReLU, Leaky ReLU, parametric ReLU, maxout
Activation functions
The sigmoid function

Squashes its input into $[0,1]$

$$y = \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{dy}{dx} = \sigma(x)(1 - \sigma(x))$$
The hyperbolic tangent (tanh) function

\[
\frac{dy}{dx} = (1 - \tanh^2(x))
\]
Pros and Cons

• Sigmoid is an historically important activation function
  – But nowadays, rarely used
• Sigmoid drawbacks
  1. It gets saturated, if the activation is close to zero or one
     • This leads to very small gradient, which disallows “transfer”ing the feedback to earlier layers
     • Initialization is also very important for this reason
  2. It is not zero-centered (not very severe)
     • Since the output is always positive, the later layers can make zig-zag (the sign of whole expression determines \( f=w^Tx+b \))
• Tanh
  – Similar to the sigmoid, it saturates
  – However, it is zero-centered.
  – Tanh is always preferred over sigmoid
  – Note: \( \text{tanh}(x) = 2 \sigma(2x) - 1 \)
Sigmoid introduces a systematic bias for the next layers (because it is always positive)

- This makes the saturation of next layer easier
- Also, the next layer has to have a bias term for good performance. Tanh doesn't have this problem b/c it's zero-centered.

Also, tanh has stronger gradients.

Compare the range of the derivatives of sigmoid and tanh.

Both functions show slow convergence.
Rectified Linear Units (ReLU)

\[
\varphi(x) = \max(0, x)
\]

\[
\frac{d \varphi(x)}{dx} = \begin{cases} 
1 & \text{if } x > 0 \\
0 & \text{otherwise}
\end{cases}
\]

**Figure 1:** A four-layer convolutional neural network with ReLUs (solid line) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons (dashed line).

[From Krizhevsky et al. (2012)]
ReLU biological motivation

There is an activation threshold under which the cell does not respond, and above which the cell responds roughly linearly with the stimulus intensity.

[From Hubel (1995), Ch 4]

A : Simple Cell

[From Crowder et al. (2007)]
ReLU: Pros and Cons

• Pros:
  – It converges much faster (claimed to be 6x faster than sigmoid/tanh)
  – Partly avoids the “vanishing gradient” problem (for x>0)
  – It is simpler to compute (simple comparison)

• Cons:
  – A ReLU neuron may “die” during training
  – A large gradient may update the weights such that the ReLU neuron may never activate again
    • Avoid large learning rate
Leaky ReLU [Maas et. al. 2014]

- $f(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x)$
  - When $x$ is negative, have a non-zero slope ($\alpha$)

- If you learn $\alpha$ during training, this is called parametric ReLU (PReLU) [He et al. 2015]
Maxout [Goodfellow et. al 2013]

- $\max(w_1^T x + b_1, w_2^T x + b_2)$

- ReLU, Leaky ReLU and PReLU are special cases of this

- Drawback: More parameters to learn!
Softplus

• A smooth approximation to the ReLU unit:
  \[ f(x) = \ln(1 + e^x) \]

• Its derivative is the sigmoid function:
  \[ f'(x) = 1/(1 + e^{-x}) \]
Activation Functions: To sum up

- Don’t use sigmoid
- If you really want, use tanh but it is worse than ReLU and its variants
- ReLU: be careful about dying neurons
- Leaky ReLU and Maxout: Worth trying

Nice visualizations by David Sheehan.
Basic algorithms to optimize MLPs (ANNs)
From the numerical optimization literature:

- Steepest descent with line search

- (Approximate) Second-order methods (i.e. Hessian is involved):
  - Newton's method
    - Exact second-order method. Approximate the cost function using a quadratic function at the current point, then take a step towards its minimum.
  - BFGS and L-BFGS
    - Approximates the inverse Hessian numerically
  - Conjugate gradient
    - Avoids the calculation of inverse Hessian by iteratively descending in conjugate directions. Don't undo progress previously made. Compared to line search: a plane is searched for a descent direction.

- Currently above methods are not common in deep learning research. Instead, more scalable and simpler methods based on stochastic gradient descent (SGD) are used.
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)]
Stochastic Gradient Descent (SGD)

It is necessary to decrease the learning rate over time. This is because the SGD gradient estimator introduces a source of noise (the random sampling of $m$ training examples) that does not vanish even when we arrive at a minimum.

Sufficient condition for convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$ 

In practice, linearly decaying learning rate upto iteration $\tau$:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha \epsilon_\tau$$

After that, rate is held constant. - Fall 2019

[From Goodfellow et al. (2016)]
SGD, setting the learning rate

• The learning rate may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time.

• This is more of an art than a science, and most guidance on this subject should be regarded with some skepticism.

• When using the linear schedule, the parameters to choose are \( \epsilon_0, \epsilon_\tau, \) and \( \tau. \)

• Usually \( \tau \) may be set to the number of iterations required to make a few hundred passes through the training set.

• Usually \( \epsilon_\tau \) should be set to roughly 1% the value of \( \epsilon_0. \)

• The main question is how to set \( \epsilon_0. \)
  
  – If it is too large, the learning curve will show violent oscillations, with the cost function often increasing significantly. Gentle oscillations are fine, especially if training with a stochastic cost function such as the cost function arising from the use of dropout.
  
  – If the learning rate is too low, learning proceeds slowly, and if the initial learning rate is too low, learning may become stuck with a high cost value.

[From Goodfellow et al. (2016)]
Adding momentum

• When there is high curvature, SGD could be slow. The method of momentum accelerates it [Polyak (1964)].

Figures from https://www.willamette.edu/~gorr/classes/cs449/momrate.html
Adding momentum

It accumulates an exponentially decaying moving average of past gradients (in $v$) and continues to move in their direction:

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left( \frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta), y^{(i)}) \right),$$

$$\theta \leftarrow \theta + v.$$ 

$\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay.
SGD with momentum

**Algorithm 8.2** Stochastic gradient descent (SGD) with momentum

**Require:** Learning rate $\epsilon$, momentum parameter $\alpha$.

**Require:** Initial parameter $\theta$, initial velocity $v$.

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: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

Compute velocity update: $v \leftarrow \alpha v - \epsilon g$

Apply update: $\theta \leftarrow \theta + v$

end while

[From Goodfellow et al. (2016)]
Without momentum (plain SGD):

\[ w^{t+1} = w^t - \eta g \quad \Rightarrow \quad w^{t+1} = w^t - \eta \|g\| u_g \]

Size of the step

With momentum:

\[ w^{t+1} = w^t - (\eta g^t - \alpha v^t) \quad \Rightarrow \quad w^{t+1} = w^t - (\eta g^t - \alpha (\alpha v^{t-1} - \eta g^{t-1})) \]

\[ \Rightarrow \quad w^{t+1} = w^t - (\eta g^t + \alpha \eta g^{t-1} - \alpha^2 v^{t-1}) \]

Size of the step now depends on how large and how aligned the subsequent gradients are.

e.g. if the gradient is always \( g \), then the method accelerates in that direction.
Online MLP demo

http://playground.tensorflow.org/
Reading assignment for next week: Chapter 8 of the “Deep Learning” book.

A basic MLP example using PyTorch:

Colab Notebook
References


• “Maxout Networks” by Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, Yoshua Bengio, 2013.

• Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2015) Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification


• Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng (2014). Rectifier Nonlinearities Improve Neural Network Acoustic Models


• Prajit Ramachandran, Barret Zoph, Quoc V. Le. “Swish: a Self-Gated Activation Function” , arxiv 10/2017