Deep Learning and Temporal Data Processing

1 - Deep Neural Networks

Andrea Palazzi

July 12, 2017

University of Modena and Reggio Emilia

Agenda

Introduction

One step back: Linear classifiers

Modeling a Neuron

Neural Networks

Training a DNN

Regularization

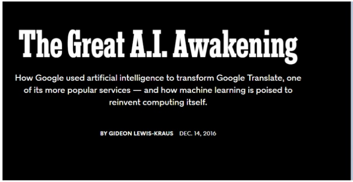
Credits

References

Introduction

Where we are now.

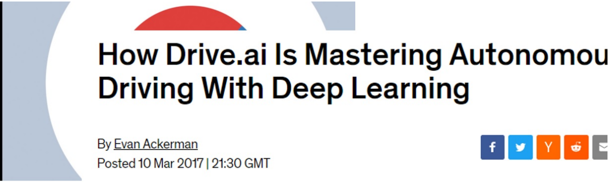
Here is the Hype



The Great A.I. Awakening

How Google used artificial intelligence to transform Google Translate, one of its more popular services — and how machine learning is poised to reinvent computing itself.

BY GIDEON LEWIS-KRAUS DEC. 14, 2016



How Drive.ai Is Mastering Autonomou Driving With Deep Learning

By Evan Ackerman
Posted 10 Mar 2017 | 21:30 GMT

WHY DEEP LEARNING IS SUDDENLY CHANGING YOUR LIFE

Deep Learning Will Radically Change the Ways We Interact with Technology

by Aditya Singh

JANUARY 30, 2017

Guardian Small

A new company every UK's AI revolution

The UK's artificial intelligence sector is boom transforming how businesses of all sizes ope

How did we get there?

Neural Networks are NOT new

Despite the current hype on the deep learning revolution, neural networks algorithms are far from being new. This is the **third time** that neural networks come to the fore:

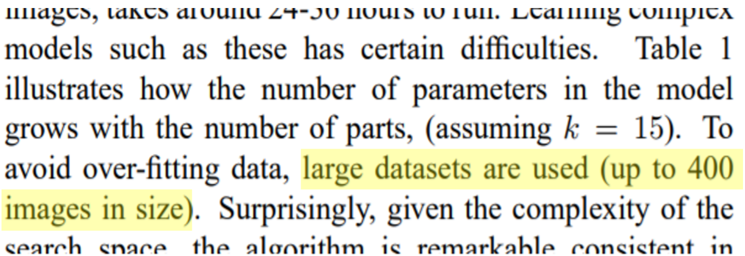
- **1940s-1960s**: theories of biological inspired learning[7, 6]. The first pioneering models such as the perceptron [8]. A single neuron is trained for the first time.
- **1980s-1990s**: neural networks with a couple of hidden layers are trained by means of backpropagation[9].
- **2006-now**: current wave of research.

Why Now?

Despite underlying ideas of deep learning have been around since a while, only recently performance boosted. This is due to a variety of complementary factors:

- large labeled datasets
- computational power
- training on GPUs (eventually distributed)
- ...

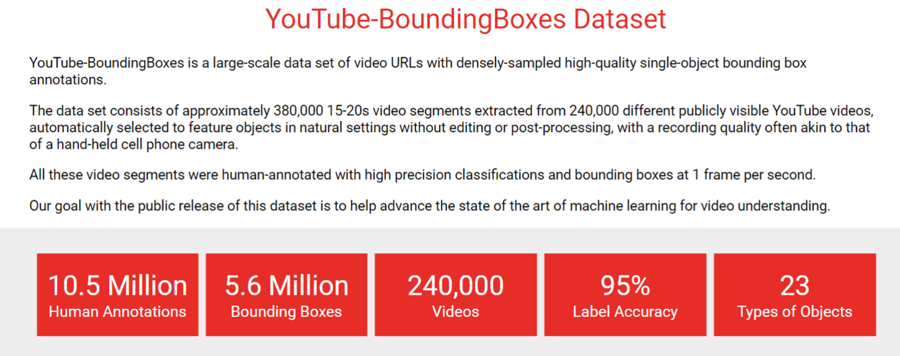
2003: Large Dataset



images, takes around 24-36 hours to run. Learning complex models such as these has certain difficulties. Table 1 illustrates how the number of parameters in the model grows with the number of parts, (assuming $k = 15$). To avoid over-fitting data, large datasets are used (up to 400 images in size). Surprisingly, given the complexity of the search space, the algorithm is remarkable consistent in

Fergus, Robert, Pietro Perona, and Andrew Zisserman. "Object class recognition by unsupervised scale-invariant learning." Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on. Vol. 2. IEEE, 2003.

2017: Large Dataset



YouTube-BoundingBoxes Dataset

YouTube-BoundingBoxes is a large-scale data set of video URLs with densely-sampled high-quality single-object bounding box annotations.

The data set consists of approximately 380,000 15-20s video segments extracted from 240,000 different publicly visible YouTube videos, automatically selected to feature objects in natural settings without editing or post-processing, with a recording quality often akin to that of a hand-held cell phone camera.

All these video segments were human-annotated with high precision classifications and bounding boxes at 1 frame per second.

Our goal with the public release of this dataset is to help advance the state of the art of machine learning for video understanding.

10.5 Million
Human Annotations

5.6 Million
Bounding Boxes

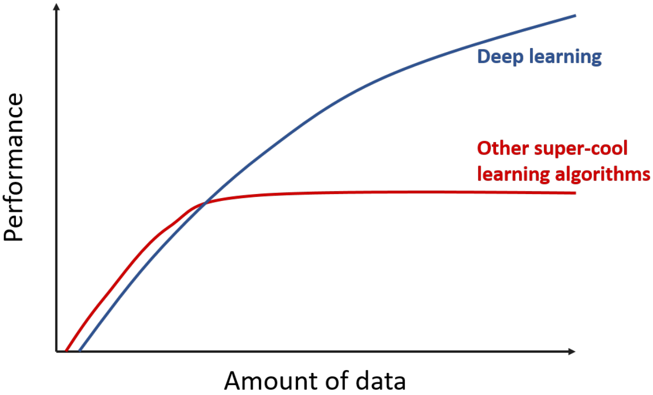
240,000
Videos

95%
Label Accuracy

23
Types of Objects

Real, Esteban, et al. "YouTube-BoundingBoxes: A Large High-Precision Human-Annotated Data Set for Object Detection in Video." arXiv preprint arXiv:1702.00824 (2017).

Why Size Matters



Andrew Ng. "What data scientists should know about deep learning".

The Challenge of Data Representation

Performance of machine learning algorithms heavily depends on the way we represent the data. These discrete pieces of information that we use as a proxy to model the complexity of the world are usually called **features**.

Generally speaking, for a long time, these features have been **handcrafted** differently for each task at hand.

Conversely, deep learning algorithms allow to **learn the right representation directly from the data**.

One step back: Linear classifiers

Linear Classifier

For the purpose of this lecture, we'll stick to the task of image classification.

Let's assume we have a training dataset of N images

$$x_i \in \mathbb{R}^D, i = 1, \dots, N$$

that we want to classify into K distinct classes.

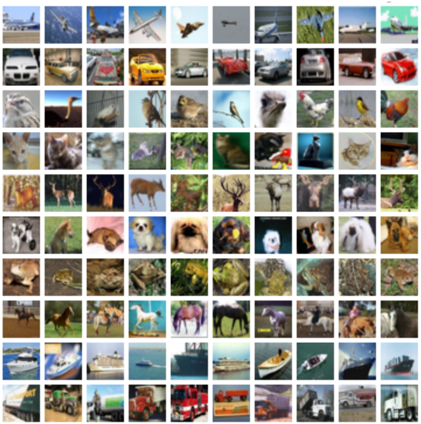
Thus, training set is made by couples:

$$(x_i, y_i), \ \ where \, y_i \in \{1, \dots, K\}$$

Our goal is to define a function $f : \mathbb{R}^D \mapsto \mathbb{R}^K$ that maps images to class scores.

Linear Classifier

Making a real-world example: let's take the `CIFAR-10` dataset, which consists of $N = 60000$ 32x32 RGB images belonging to 10 different classes.



Linear Classifier

Each image is 32x32x3, thus it can be thought as a column vector $x_i \in \mathbb{R}^{3072}$.

Now we can define a **linear mapping**:

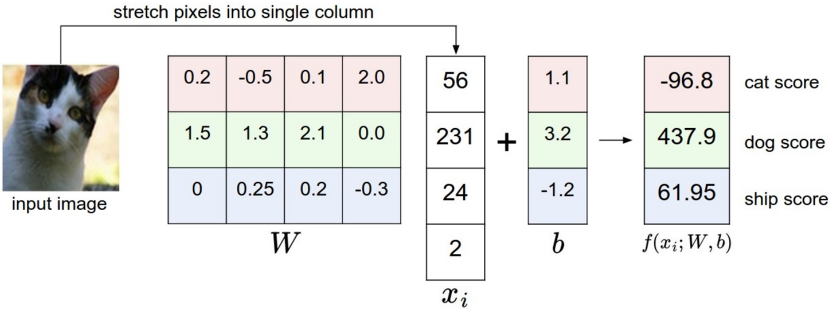
$$f(x_i, W, b) = Wx_i + b$$

where the parameters are:

- the weight matrix $W \in \mathbb{R}^{10x3072}$
- the bias vector $b \in \mathbb{R}^{10}$.

Intuitively, our goal is to learn the parameters from the training set *s.t.* when a new test image x_i^{test} is given as input, the score of the correct class is higher that the scores of other classes.

Linear Classifier



stretch pixels into single column

input image

$$\underbrace{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix}}_{W} \underbrace{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}}_{x_i} + \underbrace{\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}}_{b} \longrightarrow \underbrace{\begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix}}_{f(x_i; W, b)} \begin{matrix} \text{cat score} \\ \text{dog score} \\ \text{ship score} \end{matrix}$$

Example of mapping an image to a score. For the sake of visualization, here the image is assumed to have only 4 grayscale pixels.

Logistic Regression

If we add a sigmoid nonlinearity to our linear mapping we get a **logistic regression** classifier.

$$f(x_i, W, b) = \sigma(Wx_i + b) \qquad where\ \sigma(x) = \frac{1}{1 + e^{(-x)}}$$

which can be trained by optimizing the so called **binary cross-entropy loss**:

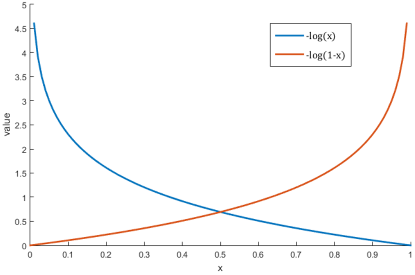
$$L(W, b) = -\frac{1}{m}\sum_{i=1}^{m}(y_i log(f_{x_i}) + (1 - y_i)log(1 - f_{x_i})) \tag{1}$$

Here m is the number of examples in the training set and f_{x_i} with an abuse of notation denotes $f(x_i, W, b)$.

Logistic Regression

Let’s try to understand better **binary cross-entropy loss**:

$$L(W, b) = -\frac{1}{m}\sum_{i=1}^{m}(y_i log(f_{x_i}) + (1 - y_i)log(1 - f_{x_i})) \tag{2}$$



Note that $y_i \in \{0, 1\}$ (binary classification task)

The label y_i act as a selector. For positive examples we penalize predictions close to zero. For negative examples we penalize predictions close to one.

Softmax Classifier

Softmax Classifier generalizes Logistic Regression classifier to multi-class classification.

We first introduce the **softmax function**:

$$softmax_j(\mathbf{z}) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

It takes a vector of arbitrary real-valued scores $\mathbf{z}$ and squashes it to a vector of values between zero and one that sum to one.

e.g.

$$\mathbf{z} = \begin{bmatrix} 1.2 \\ 5.1 \\ 2.7 \end{bmatrix} \quad softmax(\mathbf{z}) = \begin{bmatrix} 0.018 \\ 0.90 \\ 0.08 \end{bmatrix}$$

Softmax Classifier

In the softmax classifier the scores of linear function mapping $f(x_i, W) = Wx_i$ are interpreted as unnormalized log probabilities. To train the classifier we optimize the **cross-entropy loss**:

$$L = -\sum_i y_i^{true} log(y_i^{pred})$$

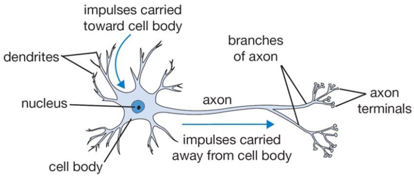
where y_i^{true} is the ground truth output distribution and y_i^{pred} is the predicted output distribution. In practice, y_i^{true} is a *one-hot* vector selecting the output of the right label.

Notice: softmax classifier has the appealing property to produce an easy-to-interpret output, that is the normalized score confidence for each class.

Modeling a Neuron

Introduction

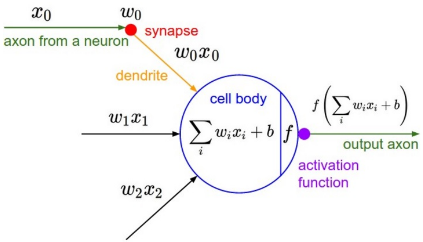
Neural Networks are a mathematical model **coarsely** inspired by the way our own brain works.


impulses carried toward cell body
branches of axon
dendrites
nucleus
axon
axon terminals
impulses carried away from cell body
cell body

Neurons are the basic computational unit of our brain. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^{14} - 10^{15} **synapses**. Each neuron receives input signals from its **dendrites** and produces output signals along its (single) **axon**. The axon eventually branches out and connects via synapses to dendrites of other neurons.

Modeling a Single Neuron

More formally, we can model a single **neuron** as follows:



Each neuron can have multiple inputs. The neuron's output is the dot product between the inputs and its weights, plus the bias: then, a non-linearity is applied.

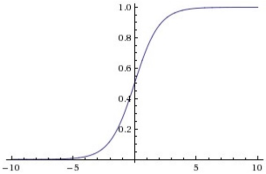
Modeling a Single Neuron

It's easy to see that a single neuron can be used to implement a binary classifier.

Indeed, when **cross-entropy loss** is applied to neuron's output, we can optimize a **binary Softmax classifier** (*a.k.a.* Logistic regression).

Neuron Activation

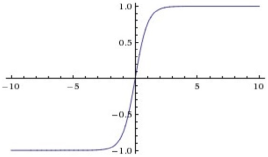
Activation functions are non-linear functions computed on the output of each neuron. There are a number of different activation functions you could use. In practice, the three most widely used functions have been *sigmoid*, *tanh* and *ReLu*. Nonetheless, more complex activation functions exist (*e.g.* [4, 3]).



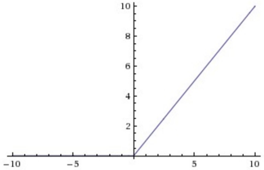
Sigmoid nonlinearity has form $\sigma(x) = 1/(1 + e^{-x})$. Sigmoid function squashes any real-valued input into range $[0, 1]$. It has seen frequent use historically, yet now it's rarely used. It has two major drawbacks:

- saturates and kill the gradient
- output is not zero centered

Neuron Activation



Tanh activation



ReLu activation

Tanh is a scaled sigmoid neuron: $tanh(x) = 2\sigma(x) - 1$. Differently to sigmoid, input is squashed in range [-1, 1], so the output is 0-centered. However, activation can still saturate and kill the gradient.

ReLu stands for Rectified Linear Unit and computes the function $f(x) = max(0, x)$. ReLu activation was found to greatly accelerate the convergence of SGD compared to sigmoid/tanh functions [5]. Furthermore, ReLu can be implemented by a simple threshold, *w.r.t.* other activations which require complex operations.

Neuron Activation

Why using non-linear activations at all?

Composition of linear functions is a linear function. Without nonlinearities, neural networks would reduce to 1 layer logistic regression.

Let's say we have the following function (ϕ represent nonlinear activations):

$$f(\boldsymbol{x}) = \phi(\mathbf{W_2}\phi(\mathbf{W_1 x}))$$

If we get rid of nonlinearities, this reduces to:

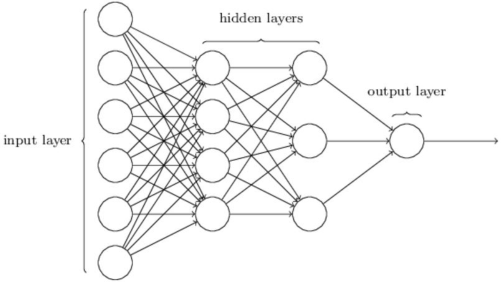
$$f(\boldsymbol{x}) = \mathbf{W_2 W_1 x} = \boldsymbol{Wx} \quad \textit{where} \quad \boldsymbol{W} = \boldsymbol{W_2 W_1}$$

which is clearly still linear.

Neural Networks

Neural Networks

When we connect an ensemble of neurons in a graph is when the magic happens and we get an actual **neural network**.



Neural networks are arranged in **layers**, with one *input layer*, one *output layer* and *N hidden layers* in the middle.

The network depicted here has a total of 47 *learnable parameters. Does this make sense to you?*

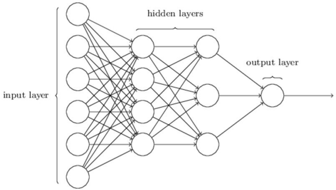
Neural Networks

The 4-layer network previously depicted can be simply expressed as:

$$out = \phi(\mathbf{W_3}\phi(\mathbf{W_2}\phi(\mathbf{W_1 x}))) \tag{3}$$

where:

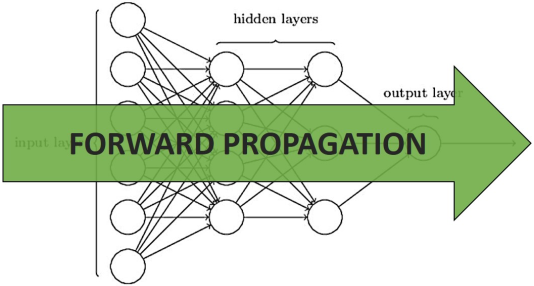
- ϕ is the activation function
- $\mathbf{x} \in \mathbb{R}^6$ is the input
- $\mathbf{W_1} \in \mathbb{R}^{4x6}$ are the weights of first layer
- $\mathbf{W_2} \in \mathbb{R}^{3x4}$ are the weights of second layer
- $\mathbf{W_3} \in \mathbb{R}^{1x3}$ are the weights of third layer



Notice that to ease the notation biases have been incorporated into weight matrices W.

Forward propagation

Forward propagation is the process of computing the network output given its input.



The formula of forward propagation for our toy network above is the one in Eq. 3.

Representational Power

It has been shown (*e.g.* [1]) that any continuous function $f(x)$ can be approximated at arbitrary precision $\epsilon > 0$ by a neural network $g(x)$ with at least one hidden layer.
That is:

$$\forall x, \quad |f(x) - g(x)| < \epsilon$$

For this reason neural networks with at least one hidden layers are referred to as **universal approximators**. In practice however networks with more layers often outperform 2-layer nets, despite the fact that on paper their representational power is equal.

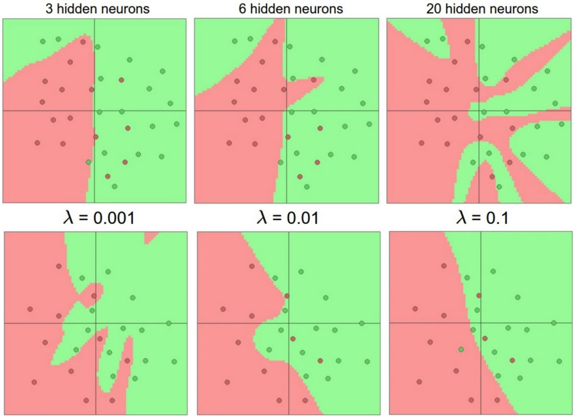
Setting Hyperparameters

The number of layers in the network, as well as the number of neurons in each layers, are so called **hyperparameters** of the architecture.

Keep in mind that:

- the bigger the size and the number of layers, the more the **capacity** of the network increases, which is good because the space of representable functions grow.
- bigger networks without proper regularization are way more prone to **overfit** the training data, which is bad.

Setting Hyperparameters



In practice, usually networks are made as big as computational budget allows.

Then overfitting is prevented through proper **regularization techniques** (*e.g.* dropout, weights decay, input noise).

We'll talk about this later.

Training a DNN

Objective Function

The **objective or loss function measures the quality of our mapping** from input to output. This function has a form of this kind:

$$L(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i L_i(\boldsymbol{\theta}; \mathbf{x_i}, \mathbf{y_i}) + \lambda \Omega(\boldsymbol{\theta})$$

where N is the number of training examples, $\boldsymbol{\theta}$ is the set of network parameters and λ weights the regularization's strength. In particular:

- The *data term*, computed as an average over individual examples, measures the goodness of model's predictions *w.r.t.* data labels.
- The *regularization term* which depends only on network parameters and has the role to mitigate the risk of overfitting.

Objective Function

The choice of the data loss depends on the task we want to solve.

- **Classification**:
  - *hinge loss*:

$$L_i = \sum_{j \neq y_i} max(0, f_j - f_{y_i} + 1) \tag{4}$$

  - *softmax loss*:

$$L_i = -log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{5}$$

- **Regression**:
  - *Mean Squared Error (MSE)*

$$L_i = ||f - y_i||_2^2 \tag{6}$$

These are just examples. The objective function is heavily task-dependent and is often customized to meet the specific problem constraints.

Learning the parameters

During training phase, **we want to learn the set of network parameters which minimize the objective function** on the training set.

More formally:

$$\boldsymbol{\theta}^* = argmin_{\theta} \left(\frac{1}{N} \sum_i L_i(\boldsymbol{\theta}; \mathbf{x_i}, \mathbf{y_i}) + \lambda \Omega(\boldsymbol{\theta}) \right)$$

Backpropagation

The algorithm used to compute the gradients of the loss functions with respect to its parameters is called **backpropagation**.

This procedure is based on *chain rule of calculus*

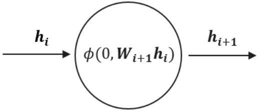
$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \quad where\ z = f(g(x)),\ y = g(x)$$

and proceeds backwards *w.r.t* the flow of computations performed to compute the loss itself (hence the name).

Backpropagation

Backpropagation is a **local** process. Neurons are completely unaware of the complete topology of the network in which they are embedded.

Indeed, in order for backpropagation to work, **each neuron just need to be able to compute two things**:

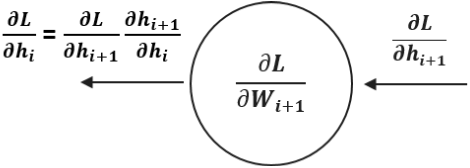


$\frac{\partial \boldsymbol{h}_{i+1}}{\partial \boldsymbol{W}_{i+1}}$ Derivative of its output with respect to its weights.

$\frac{\partial \boldsymbol{h}_{i+1}}{\partial \boldsymbol{h}_{i}}$ Derivative of its output with respect to its inputs.

Backpropagation

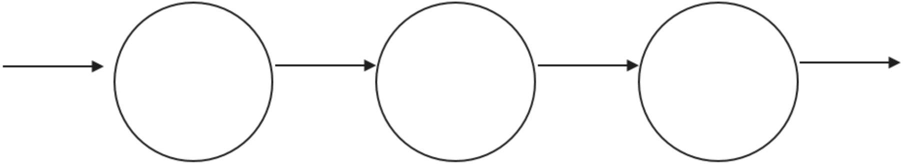
During backpropagation, the neuron will eventually discover the influence of its output value on the output of the whole network, that is will receive $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{h}_{i+1}}$.



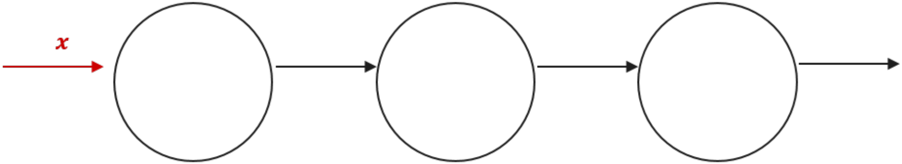
$\frac{\partial L}{\partial h_i} = \frac{\partial L}{\partial h_{i+1}} \frac{\partial h_{i+1}}{\partial h_i}$
$\frac{\partial L}{\partial W_{i+1}}$
$\frac{\partial L}{\partial h_{i+1}}$

This gradient is then chained to local gradient and passed to previous neurons to continue the backpropagation flow.

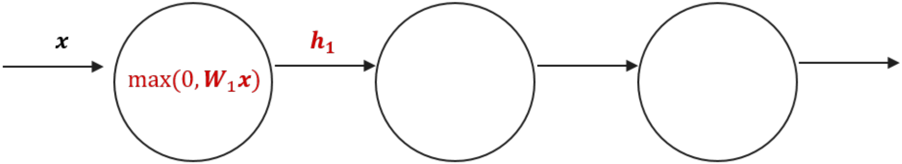
Backpropagation: example



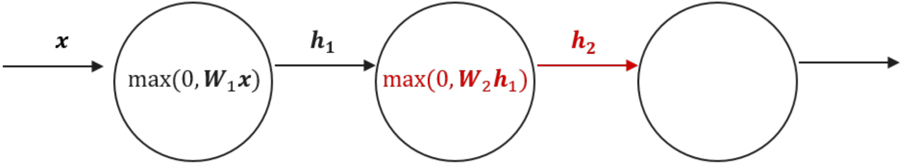
Backpropagation: example



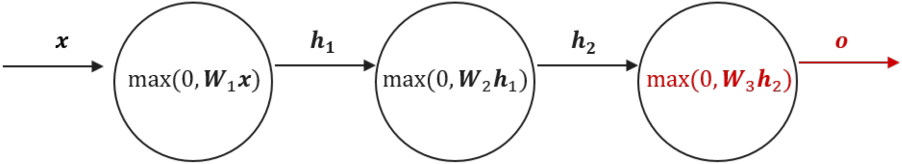
Backpropagation: example



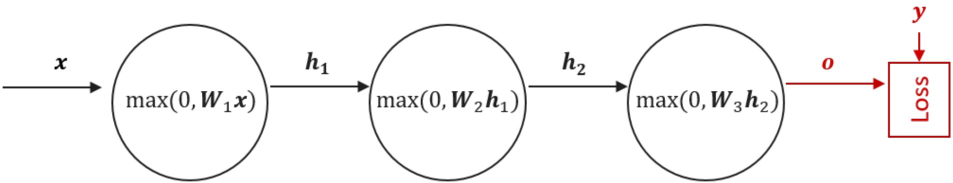
Backpropagation: example


$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1\boldsymbol{x})$
$\boldsymbol{h}_1$
$\max(0, \boldsymbol{W}_2\boldsymbol{h}_1)$
$\boldsymbol{h}_2$

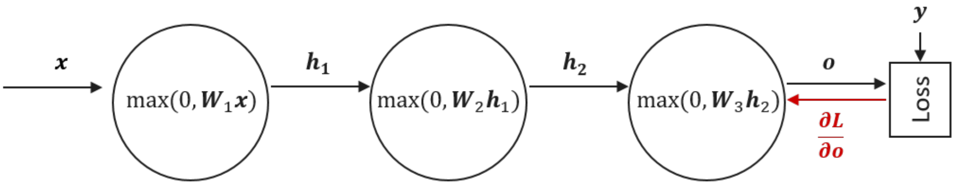
Backpropagation: example


$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1\boldsymbol{x})$
$\boldsymbol{h}_1$
$\max(0, \boldsymbol{W}_2\boldsymbol{h}_1)$
$\boldsymbol{h}_2$
$\max(0, \boldsymbol{W}_3\boldsymbol{h}_2)$
$\boldsymbol{o}$

Backpropagation: example

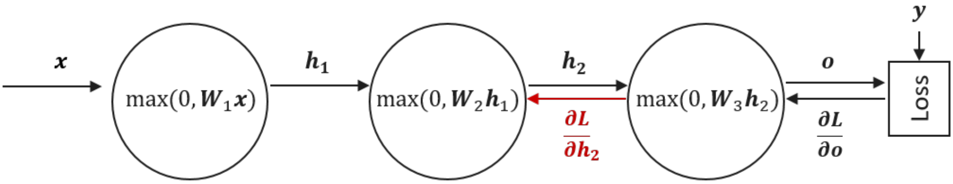

$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1\boldsymbol{x})$
$\boldsymbol{h}_1$
$\max(0, \boldsymbol{W}_2\boldsymbol{h}_1)$
$\boldsymbol{h}_2$
$\max(0, \boldsymbol{W}_3\boldsymbol{h}_2)$
$\boldsymbol{o}$
$\boldsymbol{y}$
Loss

Backpropagation: example


$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1\boldsymbol{x})$
$\boldsymbol{h}_1$
$\max(0, \boldsymbol{W}_2\boldsymbol{h}_1)$
$\boldsymbol{h}_2$
$\max(0, \boldsymbol{W}_3\boldsymbol{h}_2)$
$\boldsymbol{o}$
$\boldsymbol{y}$
Loss
$\frac{\partial L}{\partial o}$

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial W_3}$$

Backpropagation: example



$\boldsymbol{x}$

$\max(0, \boldsymbol{W}_1 \boldsymbol{x})$

$\boldsymbol{h}_1$

$\max(0, \boldsymbol{W}_2 \boldsymbol{h}_1)$

$\boldsymbol{h}_2$

$\frac{\partial L}{\partial \boldsymbol{h}_2}$

$\max(0, \boldsymbol{W}_3 \boldsymbol{h}_2)$

$\boldsymbol{o}$

$\frac{\partial L}{\partial \boldsymbol{o}}$

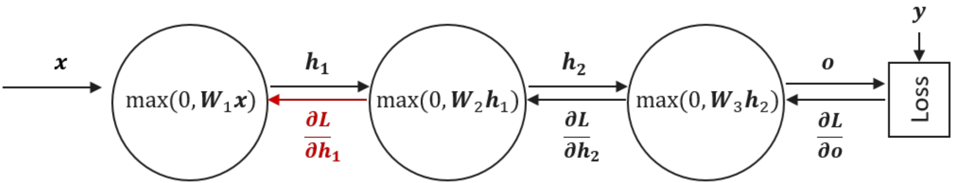
$\boldsymbol{y}$

Loss

$$\frac{\partial L}{\partial \boldsymbol{W}_2} = \frac{\partial L}{\partial \boldsymbol{h}_2} \frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{W}_2}$$

$$\frac{\partial L}{\partial \boldsymbol{W}_3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{W}_3}$$

Backpropagation: example


$\boldsymbol{x}$
$\max(0, \boldsymbol{W}_1 \boldsymbol{x})$
$\boldsymbol{h}_1$
$\frac{\partial L}{\partial h_1}$
$\max(0, \boldsymbol{W}_2 \boldsymbol{h}_1)$
$\boldsymbol{h}_2$
$\frac{\partial L}{\partial h_2}$
$\max(0, \boldsymbol{W}_3 \boldsymbol{h}_2)$
$\boldsymbol{o}$
$\frac{\partial L}{\partial o}$
$\boldsymbol{y}$
Loss

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h_1} \frac{\partial h_1}{\partial W_1}$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h_2} \frac{\partial h_2}{\partial W_2}$$

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial o} \frac{\partial o}{\partial W_3}$$

Weights Initialization

When initializing the weights in the neural network, turns out that **random initialization** is very important to **break the symmetry**.

Indeed, if all weights were initialized to the same value (*e.g.* 0) then all neurons would compute the same output, the same gradient and would eventually undergo to the same update. Clearly, this is bad.

Common practice is to initialize all weights to small random numbers centered on zero (eventually scaled by neuron's *fan-in* [2, 4]) and all biases to zero.

Regularization

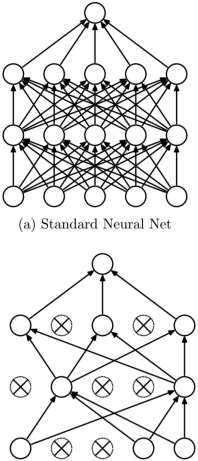
Regularizing Deep Networks

Deep networks have a **very high representational capacity**. This is good because we can learn very complex functions.

However, for this very reason bigger networks **can be prone to overfitting** issues if not regularized properly during training.

Besides traditional regularization techniques (*e.g.* L_n weight regularization), neural networks feature few simple and extremely effective regularization techniques which are commonly used in practice. We'll see them in the following slides.

Dropout



(a) Standard Neural Net

(b) After applying dropout.

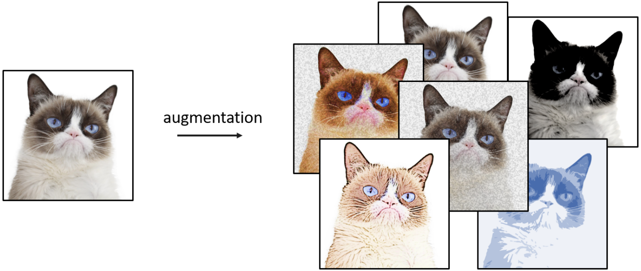
The first regularization technique which is really peculiar to neural networks is **dropout** [10].

During the training process, each neuron is set to zero (dropped) with a *drop probability* p. This can be seen as sampling at each training step a different sub-network and updating only the corresponding portion of parameters.

At testing time, no dropout is applied. This can be interpreted as taking the average prediction of the ensemble of sub-networks.

Data Augmentation

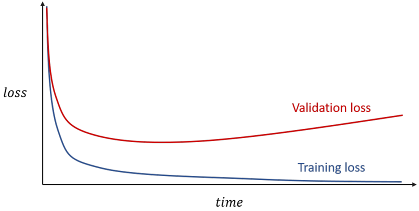
Data augmentation consists of generating new training examples by applying some transformation to $\boldsymbol{x}$ inputs in our training set. Of course, such transformation must preserve the label of the original example.



The best way to make a machine learning model generalize better would be to simply train it on more data. Unfortunately, these are not always available.

Early Stopping

The most common symptom of overfitting is that the loss on the training set steadily decreases over time, whereas it starts increasing on the validation set.



Early stopping consists of simply stopping the training process when the loss on the validation ceases to decrease. In practice, before stopping the training we may want to wait *n* epochs in which the validation loss is stable (or higher). This number of epochs is often called **patience**.

Credits

Credits i

These slides heavily borrow from a number of awesome sources. I'm really grateful to all the people who take the time to share their knowledge on this subject with others.

In particular:

- Stanford CS231n Convolutional Neural Networks for Visual Recognition
  http://cs231n.stanford.edu/
- Stanford CS20SI TensorFlow for Deep Learning Research
  http://web.stanford.edu/class/cs20si/syllabus.html
- Deep Learning Book (GoodFellow, Bengio, Courville)
  http://www.deeplearningbook.org/

Credits ii

- Marc'Aurelio Ranzato, "Large-Scale Visual Recognition with Deep Learning"
  www.cs.toronto.edu/~ranzato/publications/ranzato_cvpr13.pdf
- Convolution arithmetic animations
  https://github.com/vdumoulin/conv_arithmetic
- Andrej Karphathy personal blog
  http://karpathy.github.io/
- WildML blog on AI, DL and NLP
  http://www.wildml.com/
- Michael Nielsen Deep Learning online book
  http://neuralnetworksanddeeplearning.com/

References

References i

[1] G. Cybenko.
Approximation by superpositions of a sigmoidal function.
Mathematics of Control, Signals, and Systems (MCSS), 2(4):303–314, 1989.

[2] X. Glorot and Y. Bengio.
Understanding the difficulty of training deep feedforward neural networks.
In *Aistats*, volume 9, pages 249–256, 2010.

[3] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio.
Maxout networks.
arXiv preprint arXiv:1302.4389, 2013.

References ii

[4] K. He, X. Zhang, S. Ren, and J. Sun.
Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.
In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton.
Imagenet classification with deep convolutional neural networks.
In *Advances in neural information processing systems*, pages 1097–1105, 2012.

References iii

[6] W. S. McCulloch and W. Pitts.
A logical calculus of the ideas immanent in nervous activity.
Bulletin of mathematical biology, 5(4):115–133, 1943.

[7] R. Morris.
Do hebb: The organization of behavior, wiley: New york; 1949.
Brain research bulletin, 50(5):437, 1999.

[8] F. Rosenblatt.
The perceptron: A probabilistic model for information storage and organization in the brain.
Psychological review, 65(6):386, 1958.

References iv

[9] D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al.
Learning representations by back-propagating errors.
Cognitive modeling, 5(3):1, 1988.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov.
Dropout: A simple way to prevent neural networks from overfitting.
The Journal of Machine Learning Research, 15(1):1929–1958, 2014.