

Deep Learning and Temporal Data Processing

1 - Deep Neural Networks

Andrea Palazzi

June 21th, 2017

University of Modena and Reggio Emilia

Introduction

Linear classifiers

Modeling a Neuron

Neural Networks

Training a DNN

Credits

References

Introduction

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 GIDSON LEWIS-KRAUS DEC. 14, 2016

How Drive.ai Is Mastering Autonomous 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 20, 2017

Guardian Small

A new company every
UK's AI revolution

The UK's artificial intelligence sector is booming
transforming how businesses of all sizes operate

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**: we're here.

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)
- ...

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**.

Linear classifiers

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

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

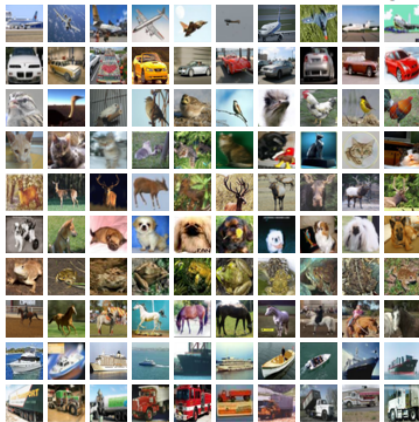
that we want to classify into K distinct classes.

Thus, training set is made by couples:

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

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

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.



Each image is $32 \times 32 \times 3$, thus it can be thought as a column vector $x_i \in \mathbb{R}^{3072}$.

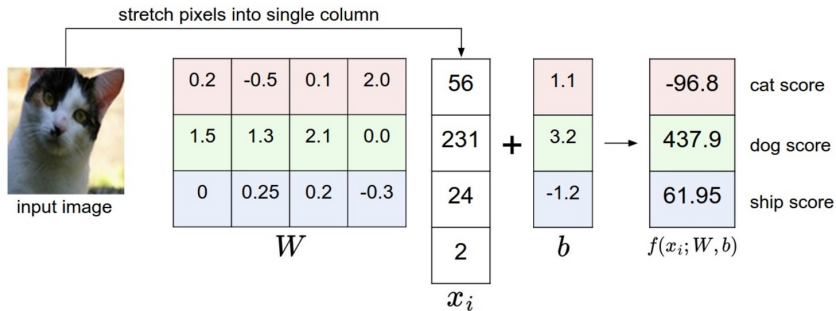
Now we can define a **linear mapping**:

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

where the parameters are:

- the weight matrix $W \in \mathbb{R}^{10 \times 3072}$
- 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 than the scores of other classes.



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

First let's introduce the **softmax function**:

$$\text{softmax}_j(\mathbf{z}) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

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.

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

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

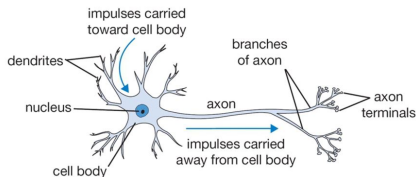
In the Softmax classifier the scores of linear function mapping $f(x_i, W) = Wx_i$ are interpreted as unnormalized log probabilities and we use the **cross-entropy loss**:

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

Prove yourself that this loss function makes sense.

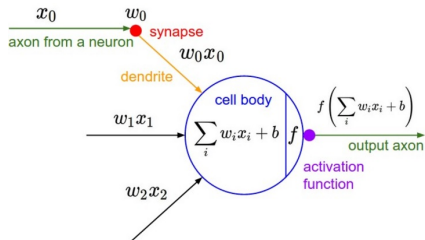
Modeling a Neuron

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



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.

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

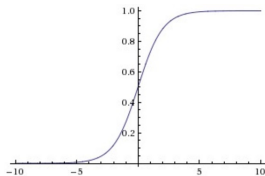


Each neuron can have multiple inputs. The neuron's output is the dot product between the inputs and its weights, plus the bias: then, a non-linearity is applied.

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

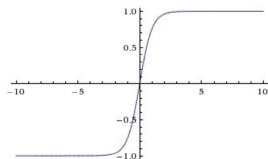
Indeed, when **cross-entropy loss** is applied to neuron's output, we can optimize a **binary Softmax classifier** (*a.k.a.* Logistic regression).

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]).

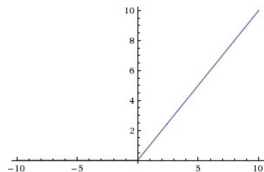


Sigmoid nonlinearity has form $\sigma(x) = 1/(1 + e^{-x})$. Sigmoid function squashes any real-valued input into range $[0, 1]$. It has seen frequent use historically, yet now it's rarely used. It has two major drawbacks:

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



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.

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):

$$f(\mathbf{x}) = \phi(\mathbf{W}_2 \phi(\mathbf{W}_1 \mathbf{x}))$$

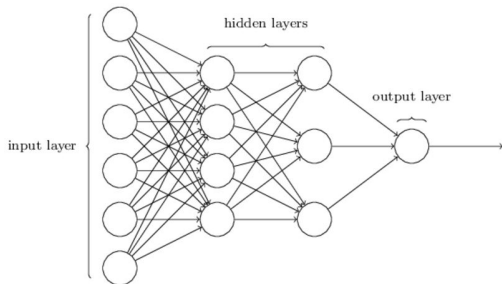
If we get rid of nonlinearities, this reduces to:

$$f(\mathbf{x}) = \mathbf{W}_2 \mathbf{W}_1 \mathbf{x} = \mathbf{W} \mathbf{x} \quad \text{where} \quad \mathbf{W} = \mathbf{W}_2 \mathbf{W}_1$$

which is clearly still linear.

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.

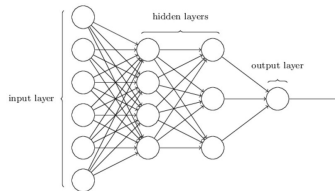
The network depicted here has a total of 47 learnable parameters. Does this make sense to you?

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

$$out = \phi(\mathbf{W}_3\phi(\mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x}))) \quad (1)$$

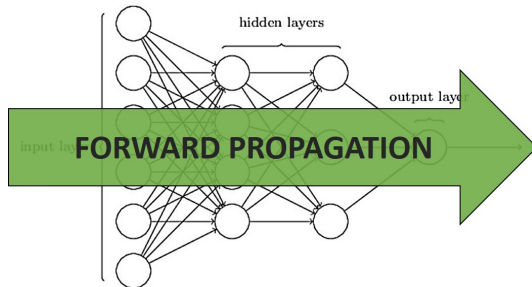
where:

- ϕ is the activation function
- $\mathbf{x} \in \mathbb{R}^6$ is the input
- $\mathbf{W}_1 \in \mathbb{R}^{4 \times 6}$ are the weights of first layer
- $\mathbf{W}_2 \in \mathbb{R}^{3 \times 4}$ are the weights of second layer
- $\mathbf{W}_3 \in \mathbb{R}^{1 \times 3}$ are the weights of third layer



Notice that to ease the notation biases have been incorporated into weight matrices \mathbf{W} .

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. 1.

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:

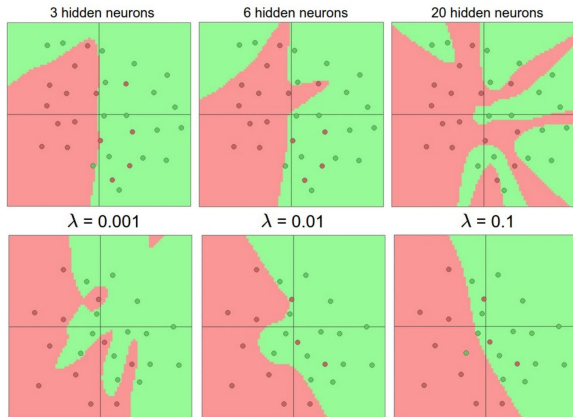
$$\forall x, \quad |f(x) - g(x)| < \epsilon$$

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.

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.



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).

Training a DNN

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

$$L(\theta; \mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i L_i(\theta; \mathbf{x}_i, \mathbf{y}_i) + \alpha \Omega(\theta)$$

where N is the number of training examples, θ is the set of network parameters and α 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.

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

- **Classification:**

- *hinge loss:*

$$L_i = \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1) \quad (2)$$

- *softmax loss:*

$$L_i = -\log \left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) \quad (3)$$

- **Regression:**

- *Mean Squared Error (MSE)*

$$L_i = \|f - y_i\|_2^2 \quad (4)$$

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

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

More formally:

$$\theta^* = \operatorname{argmin}_{\theta} \left(\frac{1}{N} \sum_i L_i(\theta; \mathbf{x}_i, \mathbf{y}_i) + \alpha \Omega(\theta) \right)$$

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*

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

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

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:**



$\frac{\partial h_{i+1}}{\partial W_{i+1}}$ Derivative of its output with respect to its weights.

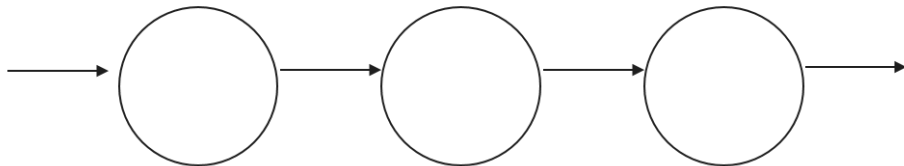
$\frac{\partial h_{i+1}}{\partial h_i}$ Derivative of its output with respect to its inputs.

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

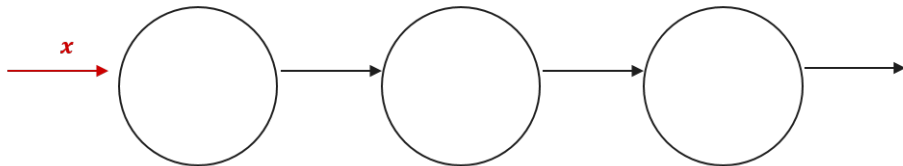


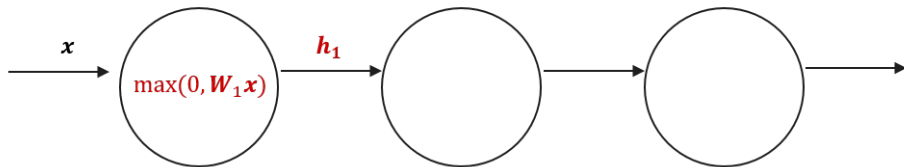
The diagram shows a central circle representing a neuron node. Inside the circle is the expression $\frac{\partial L}{\partial W_{i+1}}$. An arrow points from the right into the circle, labeled with the gradient $\frac{\partial L}{\partial h_{i+1}}$. Another arrow points from the left side of the circle, labeled with the chained gradient $\frac{\partial L}{\partial h_i} = \frac{\partial L}{\partial h_{i+1}} \frac{\partial h_{i+1}}{\partial h_i}$.

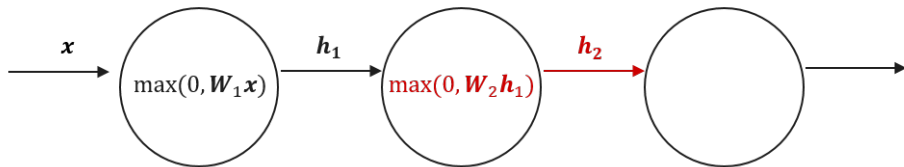
This gradient is then chained to local gradient and passed to previous neurons to continue the backpropagation flow.

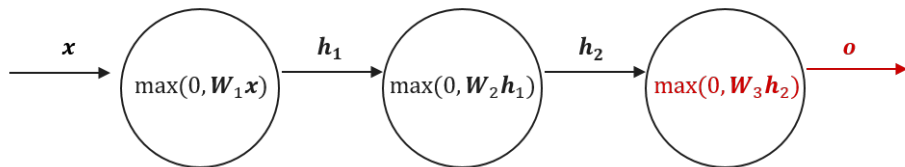


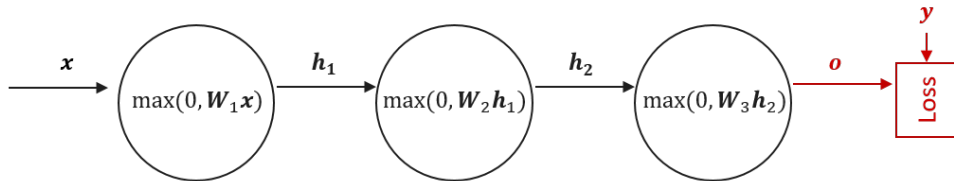
Backpropagation: example

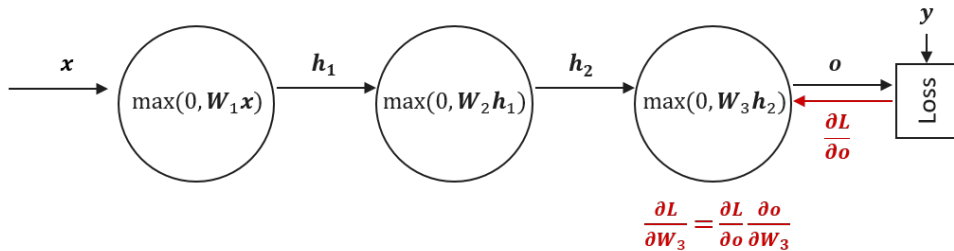


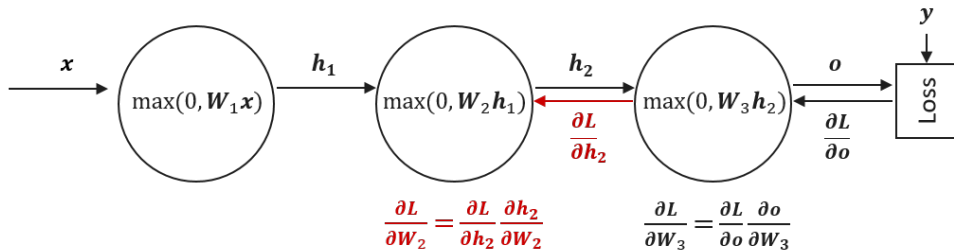


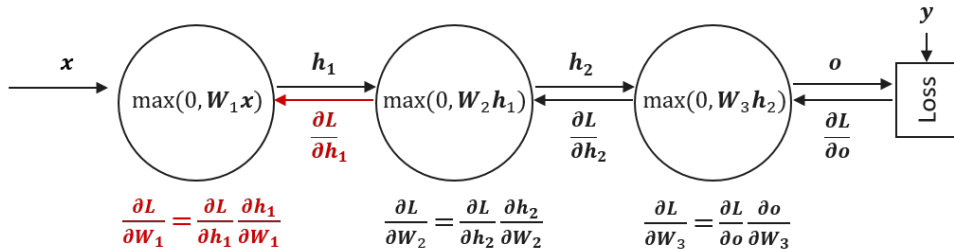








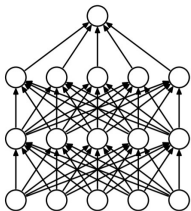




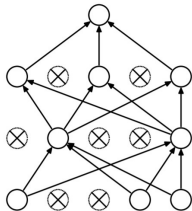
When initializing the weights in the neural network, turns out that **random initialization** is very important to **break the symmetry**.

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.



(a) Standard Neural Net



(b) After applying dropout.

Besides traditional regularization techniques (e.g. L_n weight regularization), neural networks feature a simple and extremely effective regularization which 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.

Credits

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/>
- Deep Learning Book (GoodFellow, Bengio, Courville)
<http://www.deeplearningbook.org/>
- 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

- [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.

- [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.

- [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.

- [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.