Fundamentals of Convolutional Neural Networks (CNN) Part III

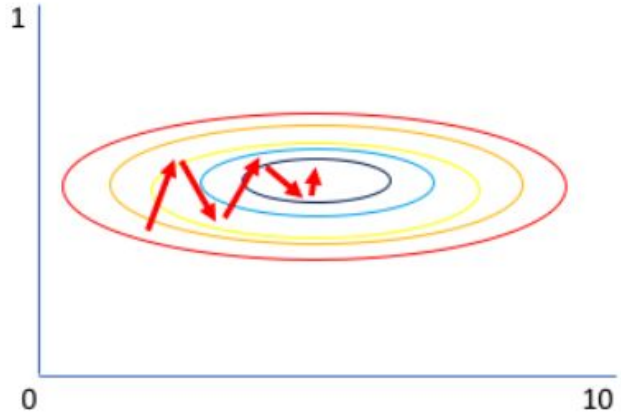
ivanovitch.silva@ufrn.br
@ivanovitchm

UFRN
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

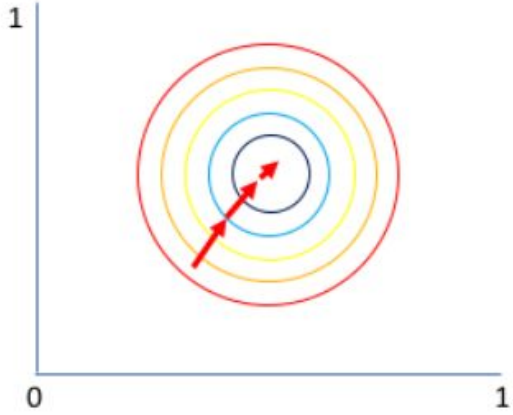
DCA

Batch Normalization

Normalizing inputs to speed up learning

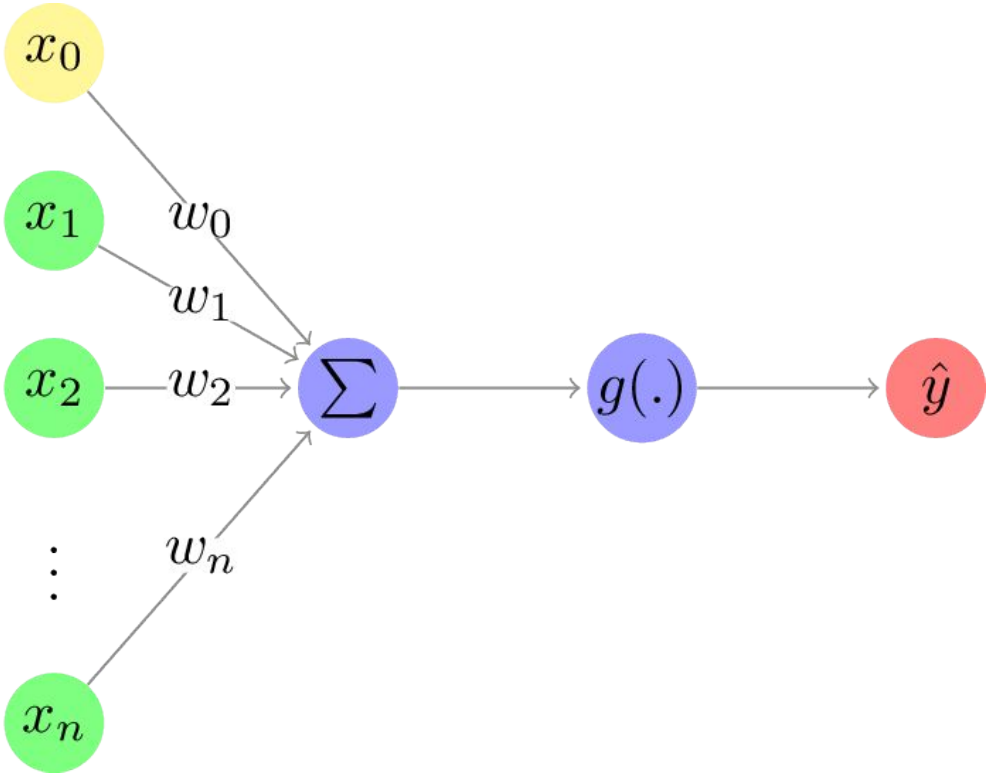


Gradient of larger parameter dominates the update



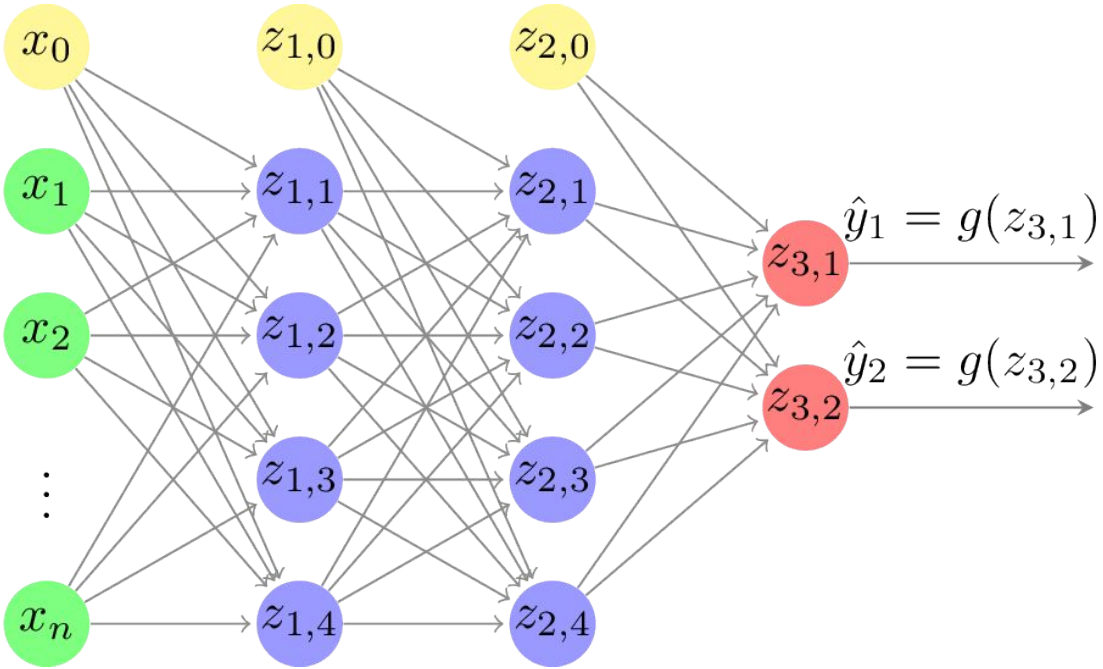
Both parameters can be updated in equal proportions

Normalizing inputs to speed up learning



$$X = \frac{X - \mu}{\sigma}$$

Normalizing inputs to speed up learning



In practice, we actually **normalize $z_{i,k}$** which has the same effect as **normalizing X.**

Batch Norm (BN)

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization *for each training mini-batch*. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout. Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

1 Introduction

Deep learning has dramatically advanced the state of the art in vision, speech, and many other areas. Stochastic

Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

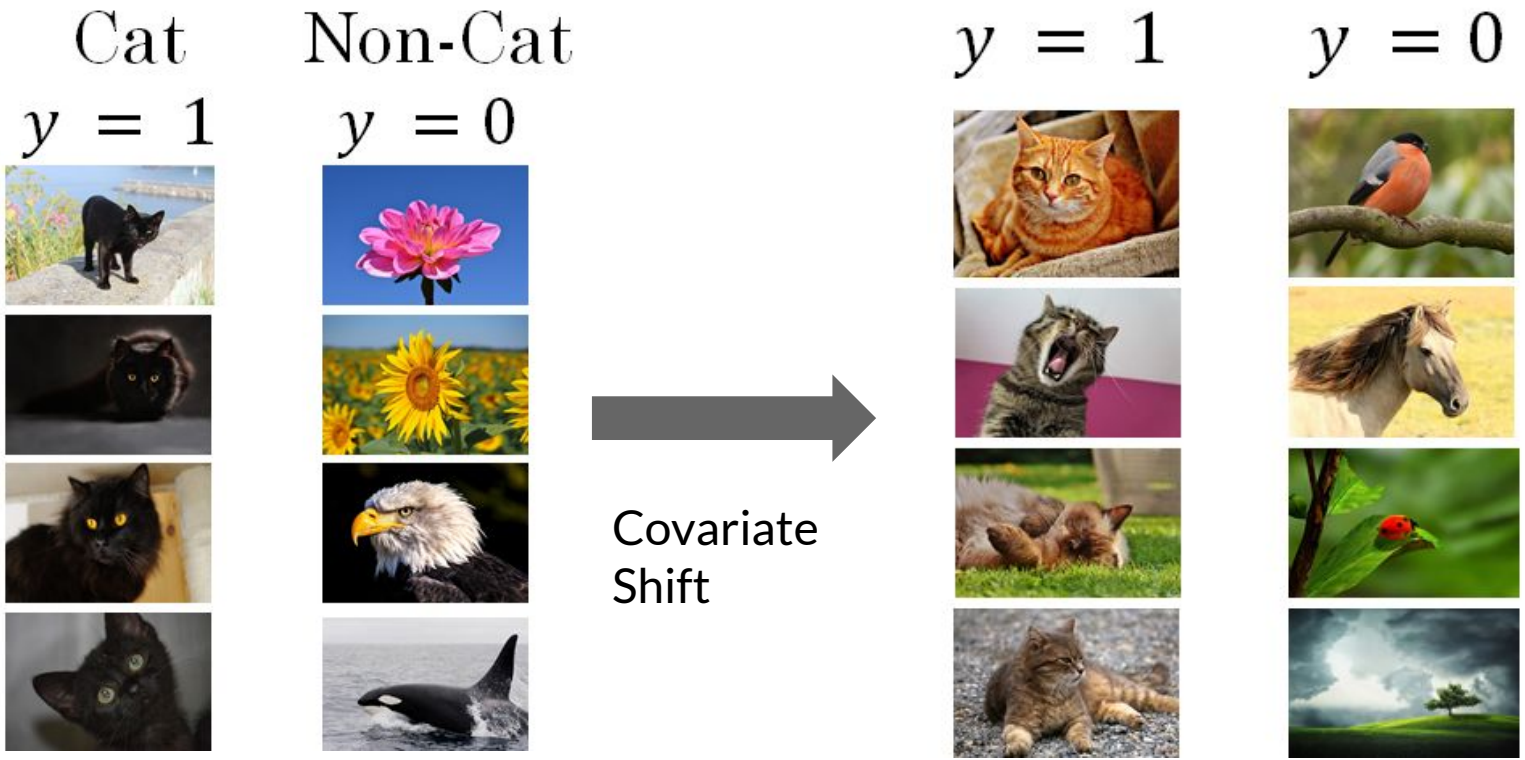
While stochastic gradient is simple and effective, it requires careful tuning of the model hyper-parameters, specifically the learning rate used in optimization, as well as the initial values for the model parameters. The training is complicated by the fact that the inputs to each layer are affected by the parameters of all preceding layers – so that small changes to the network parameters amplify as the network becomes deeper.

The change in the distributions of layers' inputs presents a problem because the layers need to continuously adapt to the new distribution. When the input distribution to a learning system changes, it is said to experience *covariate shift* (Shimodaira, 2000). This is typically handled via domain adaptation (Jiang, 2008). However, the notion of covariate shift can be extended beyond the learning system as a whole, to apply to its parts, such as a sub-network or a layer. Consider a network computing

$$\ell = F_2(F_1(\mathrm{u}, \Theta_1), \Theta_2)$$

where F_1 and F_2 are arbitrary transformations, and the parameters Θ_1, Θ_2 are to be learned so as to minimize

Learning on Shifting Input Distribution



How Does Batch Normalization Help Optimization?

Shibani Santurkar* MIT shibani@mit.edu
Dimitris Tsipras* MIT tsipras@mit.edu
Andrew Ilyas* MIT ailyas@mit.edu
Aleksander Mądry MIT madry@mit.edu

Abstract

Batch Normalization (BatchNorm) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). Despite its pervasiveness, the exact reasons for BatchNorm's effectiveness are still poorly understood. The popular belief is that this effectiveness stems from controlling the change of the layers' input distributions during training to reduce the so-called "internal covariate shift". In this work, we demonstrate that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, we uncover a more fundamental impact of BatchNorm on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, allowing for faster training.

1 Introduction

Over the last decade, deep learning has made impressive progress on a variety of notoriously difficult tasks in computer vision [16, 7], speech recognition [5], machine translation [29], and game-playing [18, 25]. This progress hinged on a number of major advances in terms of hardware, datasets [15, 23], and algorithmic and architectural techniques [27, 12, 20, 28]. One of the most prominent examples of such advances was batch normalization (BatchNorm) [10].

At a high level, BatchNorm is a technique that aims to improve the training of neural networks by stabilizing the distributions of layer inputs. This is achieved by introducing additional network layers

Published as a conference paper at ICLR 2021

DECONSTRUCTING THE REGULARIZATION OF BATCHNORM

Yann N. Dauphin
Google Research
ynd@google.com

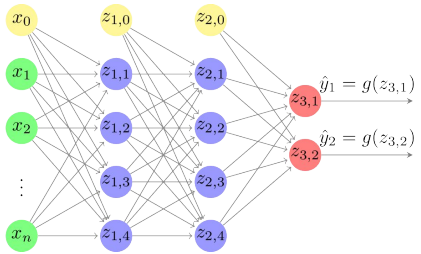
Ekin D. Cubuk
Google Research
cubuk@google.com

ABSTRACT

Batch normalization (BatchNorm) has become a standard technique in deep learning. Its popularity is in no small part due to its often positive effect on generalization. Despite this success, the regularization effect of the technique is still poorly understood. This study aims to decompose BatchNorm into separate mechanisms that are much simpler. We identify three effects of BatchNorm and assess their impact directly with ablations and interventions. Our experiments show that preventing explosive growth at the final layer at initialization and during training can recover a large part of BatchNorm's generalization boost. This regularization mechanism can lift accuracy by 2.9% for Resnet-50 on Imagenet without BatchNorm. We show it is linked to other methods like Dropout and recent initializations like Fixup. Surprisingly, this simple mechanism matches the improvement of 0.9% of the more complex Dropout regularization for the state-of-the-art Efficientnet-B8 model on Imagenet. This demonstrates the underrated effectiveness of simple regularizations and sheds light on directions to further improve generalization for deep nets.

Implementing BN in a single layer

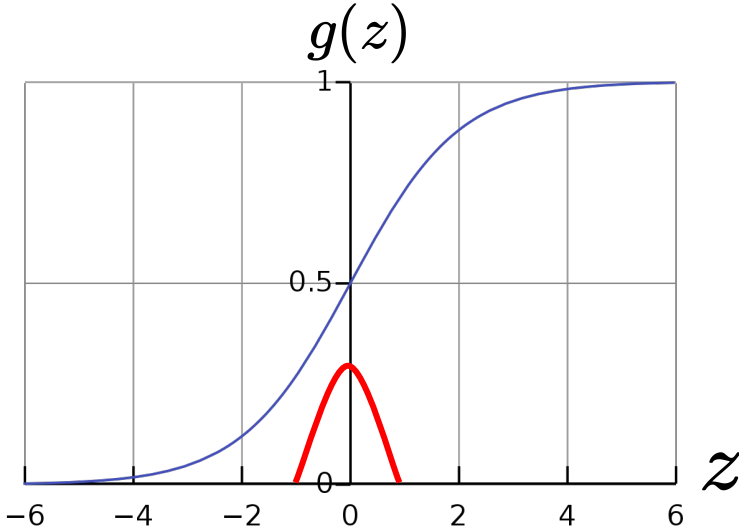
$$All \;\; z_l = \{z_l^{(1)}, z_l^{(2)}, \ldots, z_l^{(m)}\}$$



$$\mu = \frac{1}{m}\sum_i z_l^{(i)}$$

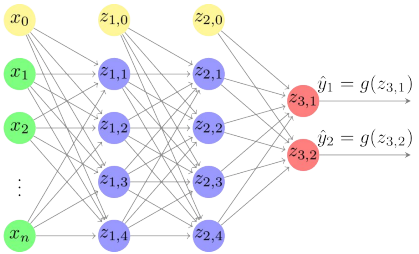
$$\sigma^2 = \frac{1}{m}\sum_i (z_l^{(i)} - \mu)^2$$

$$z_{l\_norm}^{(i)} = \frac{z_l^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$



Implementing BN in a single layer

$$All \;\; z_l = \{z_l^{(1)}, z_l^{(2)}, \ldots, z_l^{(m)}\}$$



$$\mu = \frac{1}{m}\sum_i z_l^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_l^{(i)} - \mu)^2$$

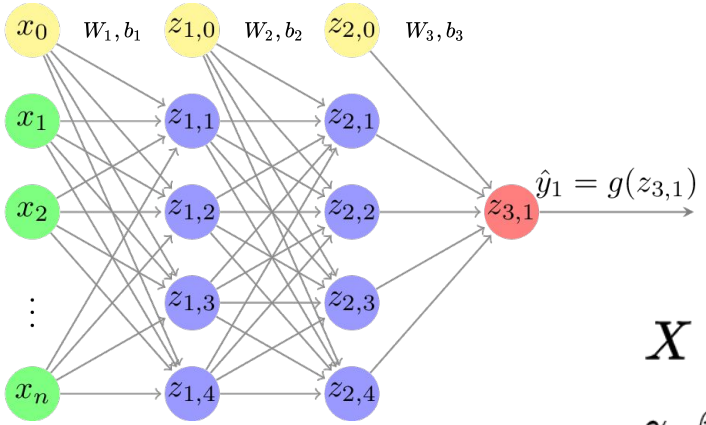
$$z_{l\_norm}^{(i)} = \frac{z_l^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\tilde{z}_l^{(i)} = \gamma \, z_{l\_norm}^{(i)} + \beta$$

$$if \;\; \gamma = \sqrt{\sigma^2 + \epsilon} \;\; and \;\; \beta = \mu$$
$$z_{l\_norm}^{(i)} = z_l^{(i)}$$

- γ and β are learnable parameters (gradient descent)
- γ and β are used to avoid all layers have activations with mean zero and standard deviation equal to one.

Adding BN to a Network



$$X \xrightarrow{W_1,b_1} Z_1 \xrightarrow[\text{BN}]{\gamma_1,\beta_1} \tilde{Z}_1 \rightarrow a_1 = g_1(\tilde{Z}_1) \xrightarrow{W_2,b_2} Z_2$$

$$\xrightarrow[\text{BN}]{\gamma_2,\beta_2} \tilde{Z}_2 \rightarrow a_2 = g_2(\tilde{Z}_2) \xrightarrow{W^{[3]},b^{[3]}} \dots$$

Parameters (learnable from Gradient Descent)

$$\begin{cases} W_1, b_1, W_2, b_2, W_3, b_3 \\ \gamma_1, \beta_1, \gamma_2, \beta_2, \gamma_3, \beta_3 \end{cases}$$

Working with Mini-Batches

$$X^{\{1\}} \xrightarrow{W_1,b_1} Z_1 \xrightarrow[\text{BN}]{\gamma_1,\beta_1} \tilde{Z}_1 \rightarrow a_1 = g_1(\tilde{Z}_1) \xrightarrow{W_2,b_2} Z_2$$

$$\xrightarrow[\text{BN}]{\gamma_2,\beta_2} \tilde{Z}_2 \rightarrow a_2 = g_2(\tilde{Z}_2) \xrightarrow{W^{[3]},b^{[3]}} \dots$$

$$\vdots$$

$$X^{\{t\}} \dots$$

BN calculates the normalization using data from individual batch t

$$\tilde{z}_l^{\{i\}} = \gamma\, z_{l\_norm}^{\{i\}} + \beta$$

Batch Normalization as Regularization

- Batch norm has **slight regularization effect** because it kind of **can cancel out large W**, adding noise.
  - Each mini-batch is scaled by the mean/standard deviation computed on just that mini-batch. This adds some noise to the values Z_l within that mini-batch. So similar to dropout, it adds some noise to each hidden layer activations.
- Its **regularization effect gets weaker as batch size gets larger** because there is less noise as batch size gets larger

```
# Batch Normalization Before Activation Function [Original Paper]
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(25, kernel_initializer="he_uniform"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation(tf.nn.relu))

model.add(tf.keras.layers.Dense(12,kernel_initializer="he_uniform"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation(tf.nn.relu))
```

```
# Batch Normalization After Activation Function
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(25, activation=tf.nn.relu,kernel_initializer="he_uniform"))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Dense(12, activation=tf.nn.relu,kernel_initializer="he_uniform"))
model.add(tf.keras.layers.BatchNormalization())
```

```
# Batch Normalization used to standardize the raw input variables
model = tf.keras.Sequential()
model.add(BatchNormalization(input_shape=(2,)))
```

Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

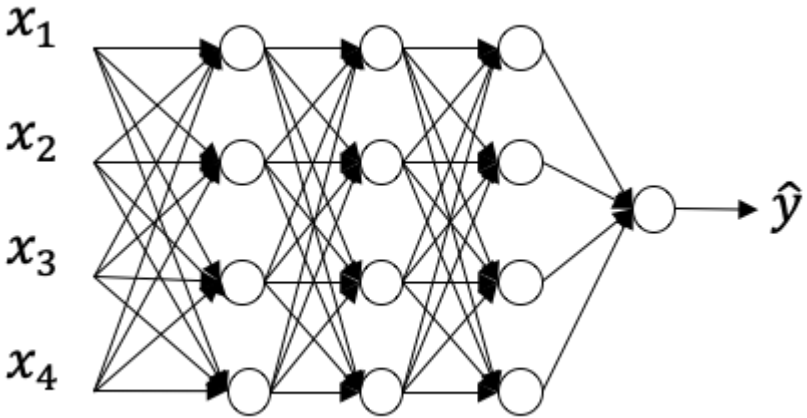
Editor: Yoshua Bengio

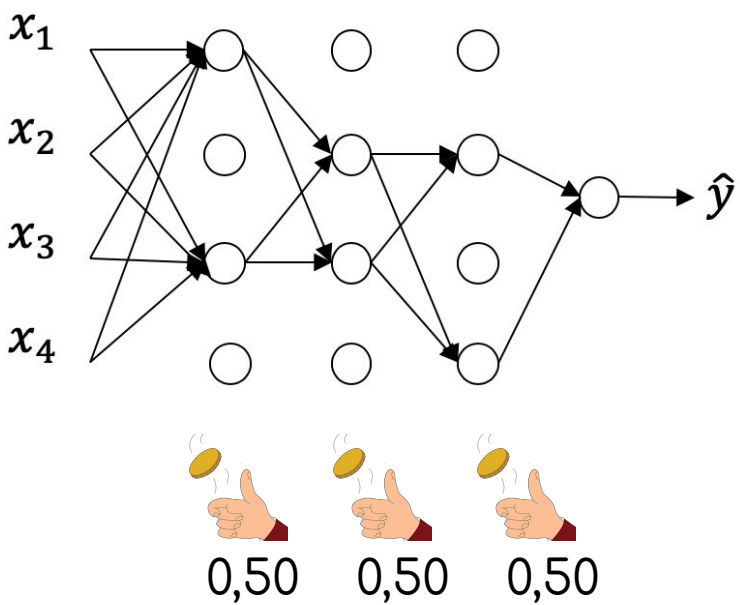
Abstract

Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. We show that dropout improves the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification and computational biology, obtaining state-of-the-art results on many benchmark data sets.

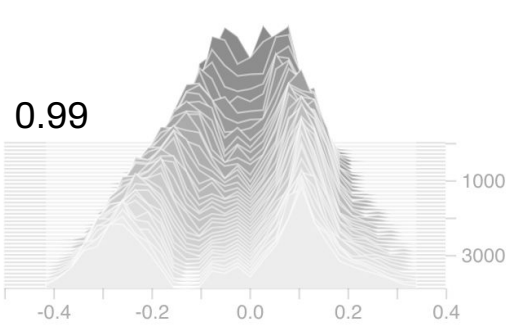
Dropout Regularization

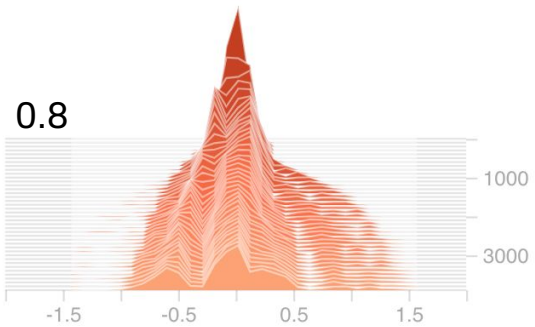
You implement Dropout regularization only while training the network. You do not apply it while running your testing data through it as you do not want any randomness in your predictions

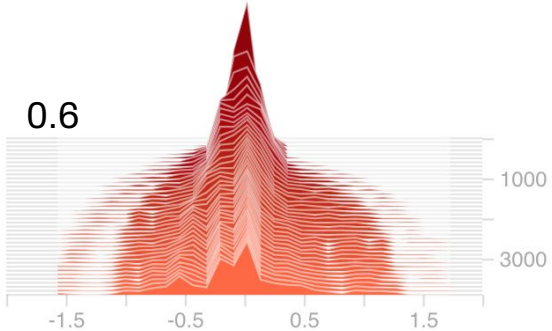


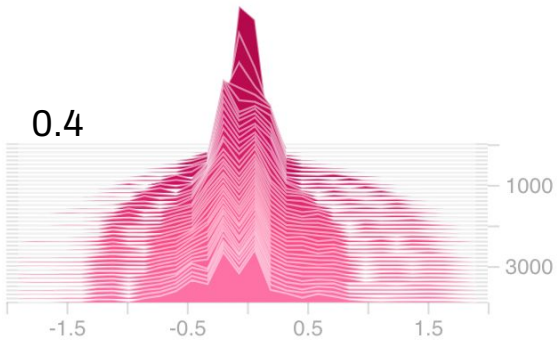


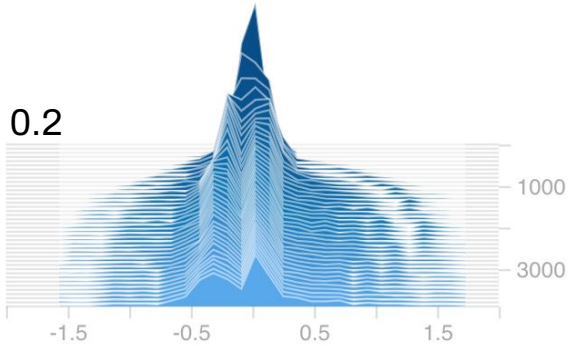
Dropout Regularization

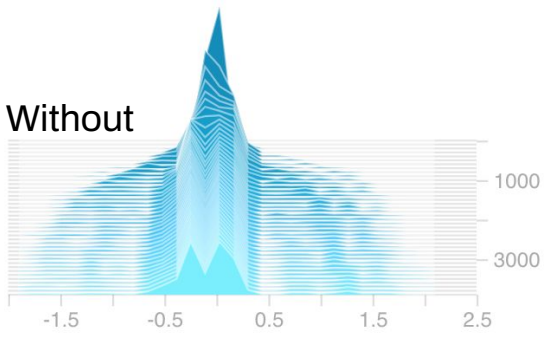












Implementing Dropout ("Inverted Dropout")

Illustrate with layer "l" = 3

$$\begin{cases} keep\_prob & = 0.8 \\ 1 - keep\_prob & = 0.2 \end{cases}$$

$$d3 = np.randon.rand(a3.shape[0], a3.shape[1]) < keep\_prob$$

$$a3 = np.multiply(a3, d3)$$

$$a3 /= keep\_prob$$

$$Z^{[4]} = \alpha^{[3]} W^{[4]} + b^{[4]}$$

It is necessary not to impact the value of Z.

Applying dropout for a input

```
import tensorflow as tf
import numpy as np

tf.random.set_seed(0)
layer = tf.keras.layers.Dropout(.2, input_shape=(2,))
data = np.arange(10).reshape(5, 2).astype(np.float32)
print(data)

[[0. 1.]
 [2. 3.]
 [4. 5.]
 [6. 7.]
 [8. 9.]]
```

keep_prob=0.8

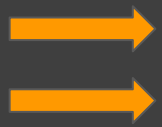
```
outputs = layer(data, training=True)
print(outputs)

tf.Tensor(
[[ 0.    1.25]
 [ 2.5   3.75]
 [ 5.    6.25]
 [ 7.5   8.75]
 [10.    0.  ]], shape=(5, 2), dtype=float32)
```

output = output /keep_prob

Putting it all together

```
model_dropout = tf.keras.Sequential([
                            tf.keras.layers.Dense(20, activation=tf.nn.relu),
                            tf.keras.layers.Dropout(0.3),
                            tf.keras.layers.Dense(3, activation=tf.nn.relu),
                            tf.keras.layers.Dropout(0.3),
                            tf.keras.layers.Dense(1, activation = tf.nn.sigmoid)
                            ])
```



Rule of the thumb

>> You implement Dropout regularization only while training the network.

>> You do not apply it while running your testing data through it as you do not want any randomness in your predictions.