1. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?

No matter what was the input - if all weights are the same, all units in hidden layer will be the same too. This is the main issue with symmetry and reason why you should initialize weights randomly (or, at least, with different values). Note, that this issue affects all architectures that use each-to-each connections.

The weights of artificial neural networks must be initialized to small random numbers.

This is because this is an expectation of the stochastic optimization algorithm used to train the model, called stochastic gradient descent.

To understand this approach to problem solving, you must first understand the role of nondeterministic and randomized algorithms as well as the need for stochastic optimization algorithms to harness randomness in their search process.

In this post, you will discover the full background as to why neural network weights must be randomly initialized.

After reading this post, you will know:

About the need for nondeterministic and randomized algorithms for challenging problems.

The use of randomness during initialization and search in stochastic optimization algorithms.

That stochastic gradient descent is a stochastic optimization algorithm and requires the random initialization of network weights.

Kick-start your project with my new book [Deep Learning With Python](https://machinelearningmastery.com/deep-learning-with-python/), including step-by-step tutorials and the Python source code files for all examples.

Let’s get started.

2.Is it OK to initialize the bias terms to 0?

Zero Initialization (Initialized all weights to 0)

It is important to note that setting biases to 0 will not create any problems as non-zero weights take care of breaking the symmetry and even if bias is 0, the values in every neuron will still be different.

Building even a simple neural network can be a confusing task and tuning it to get the better result is an extremely tedious task. The most common problem with Deep Neural Networks is Vanishing and Exploding gradient descent. To solve these issues, one solution could be to initialize the parameters carefully. In this article, we will discuss Weight initialization techniques.

This article has been written under the assumption that you have a basic understanding of neural networks, weights, biases, activation functions, forward and backward propagation, etc.

Table of Contents

 Basics and notations of neural networks

 Steps of training a neural network

 Why weight initialization?

 Different Weight initialization techniques

 Best practices of weight initialization

 Conclusion

Basics and Notations

Consider a neural network having an l layer, which has l-1 hidden layers and 1 output layer. Then, the parameters i.e, weights and biases of the layer l are represented as,

100+ Data Science Job Openings

Lenovo, TVS, Convergytics, Ripik.AI and many more are hiring | Open to all Data Science Enthusiasts.[Register Now](https://datahack.analyticsvidhya.com/jobathon/?utm_source=blog_india&utm_medium=desktop_banner_between_articles&utm_campaign=27-Oct-2022||&utm_content=registrations)

Image Source: [link](https://cdn-images-1.medium.com/max/800/1*UbPOjOD6IATWM2VDMA3JBg.png)

In addition to weights and biases, some more intermediate variables are also computed during the training process,

Image Source:[link](https://cdn-images-1.medium.com/max/800/1*xTKaL0jOvwfOso90ZOGa1Q.png)

Steps of Training a Neural Network

Training a neural network consists of the following basic steps:

Step-1: Initialization of Neural Network: Initialize weights and biases.

Step-2: Forward propagation: Using the given input X, weights W, and biases b, for every layer we compute a linear combination of inputs and weights (Z)and then apply activation function to linear combination (A). At the final layer, we compute f(A(l-1)) which could be a sigmoid (for binary classification problem), softmax (for multi-class classification problem), and this gives the prediction y\_hat.

Step-3: Compute the loss function: The loss function includes both the actual label y and predicted label y\_hat in its expression. It shows how far our predictions from the actual target, and our main objective is to minimize the loss function.

Step-4: Backward Propagation: In backpropagation, we find the gradients of the loss function, which is a function of y and y\_hat, and gradients wrt A, W, and b called dA, dW, and db. By using these gradients, we update the values of the parameters from the last layer to the first layer.

Step-5: Repeat steps 2–4 for n epochs till we observe that the loss function is minimized, without overfitting the train data.

For Example,

For a neural network having 2 layers, i.e. one hidden layer. (Here bias term is not added just for the simplicity)

Image Source:[link](https://cdn-images-1.medium.com/max/1000/1*XWlcMkDcosYwIAcTO2cCQw.png)

Image Source:[link](https://cdn-images-1.medium.com/max/1000/1*5A9bmIhGt_Rx8piZ_ybtsg.png)

Why Weight Initialization?

Its main objective is to prevent layer activation outputs from exploding or vanishing gradients during the forward propagation. If either of the problems occurs, loss gradients will either be too large or too small, and the network will take more time to converge if it is even able to do so at all.

If we initialized the weights correctly, then our objective i.e, optimization of loss function will be achieved in the least time otherwise converging to a minimum using gradient descent will be impossible.

Different Weight Initialization Techniques

One of the important things which we have to keep in mind while building your neural network is to initialize your weight matrix for different connections between layers correctly.

Let us see the following two initialization scenarios which can cause issues while we training the model:

Zero Initialization (Initialized all weights to 0)

If we initialized all the weights with 0, then what happens is that the derivative wrt loss function is the same for every weight in W[l], thus all weights have the same value in subsequent iterations. This makes hidden layers symmetric and this process continues for all the n iterations. Thus initialized weights with zero make your network no better than a linear model. It is important to note that setting biases to 0 will not create any problems as non-zero weights take care of breaking the symmetry and even if bias is 0, the values in every neuron will still be different.

Random Initialization (Initialized weights randomly)

– This technique tries to address the problems of zero initialization since it prevents neurons from learning the same features of their inputs since our goal is to make each neuron learn different functions of its input and this technique gives much better accuracy than zero initialization.

– In general, it is used to break the symmetry. It is better to assign random values except 0 to weights.

– Remember, neural networks are very sensitive and prone to overfitting as it quickly memorizes the training data.

Now, after reading this technique a new question comes to mind: “What happens if the weights initialized randomly can be very high or very low?”

(a) Vanishing gradients :

 For any activation function, abs(dW) will get smaller and smaller as we go backward with every layer during backpropagation especially in the case of deep neural networks. So, in this case, the earlier layers’ weights are adjusted slowly.

Due to this, the weight update is minor which results in slower convergence.

This makes the optimization of our loss function slow. It might be possible in the worst case, this may completely stop the neural network from training further.

More specifically, in the case of the sigmoid and tanh and activation functions, if your weights are very large, then the gradient will be vanishingly small, effectively preventing the weights from changing their value. This is because abs(dW) will increase very slightly or possibly get smaller and smaller after the completion of every iteration.

So, here comes the use of the RELU activation function in which vanishing gradients are generally not a problem as the gradient is 0 for negative (and zero) values of inputs and 1 for positive values of inputs.

(b) Exploding gradients :

This is the exact opposite case of the vanishing gradients, which we discussed above.

Consider we have weights that are non-negative, large, and having small activations A. When these weights are multiplied along with the different layers, they cause a very large change in the value of the overall gradient (cost). This means that the changes in W, given by the equation W= W — ⍺ \* dW, will be in huge steps, the downward moment will increase.

Problems occurred due to exploding gradients:

– This problem might result in the oscillation of the optimizer around the minima or even overshooting the optimum again and again and the model will never learn!

– Due to the large values of the gradients, it may cause numbers to overflow which results in incorrect computations or introductions of NaN’s (missing values).

Best Practices for Weight Initialization

 Use RELU or leaky RELU as the activation function, as they both are relatively robust to the vanishing or exploding gradient problems (especially for networks that are not too deep). In the case of leaky RELU, they never have zero gradients. Thus they never die and training continues.

 Use Heuristics for weight initialization: For deep neural networks, we can use any of the following heuristics to initialize the weights depending on the chosen non-linear activation function.

While these heuristics do not completely solve the exploding or vanishing gradients problems, they help to reduce it to a great extent. The most common heuristics are as follows:

(a) For RELU activation function: This heuristic is called He-et-al Initialization.

In this heuristic, we multiplied the randomly generated values of W by:

Image Source: [link](https://cdn-images-1.medium.com/max/800/1*-vY3G0W-4nJo-dQ1jm0p0w.png)

(b) For tanh activation function : This heuristic is known as Xavier initialization.

In this heuristic, we multiplied the randomly generated values of W by:

Image Source: [link](https://cdn-images-1.medium.com/max/800/1*El7FG2KM4zMRCV9w7diFTg.png)

(c) Another commonly used heuristic is:

Image Source: [link](https://cdn-images-1.medium.com/max/800/1*R6uFrz9qndzruz5yTRhaAA.png)

Benefits of using these heuristics:

All these heuristics serve as good starting points for weight initialization and they reduce the chances of exploding or vanishing gradients.

All these heuristics do not vanish or explode too quickly, as the weights are neither too much bigger than 1 nor too much less than 1.

They help to avoid slow convergence and ensure that we do not keep oscillating off the minima.

 Gradient Clipping:  It is another way for dealing with the exploding gradient problem. In this technique, we set a threshold value, and if our chosen function of a gradient is larger than this threshold, then we set it to another value.

3. Name three advantages of the SELU activation function over ReLU.

Like ReLU, SELU does not have vanishing gradient problem and hence, is used in deep neural networks. Compared to ReLUs, SELUs cannot die. SELUs learn faster and better than other activation functions without needing further procession.

Background on SELU

Normalized outputs seem to be really helpful in stabilizing the training process. That’s the main reason behind the popularity of [BatchNormalization](https://arxiv.org/abs/1502.03167). [SELU](https://arxiv.org/abs/1706.02515) is a way to output the normalized activations to the next layer.

The overall function is really simple:

For mean 0 and stdev 1 inputs, the values of α and λ come out to be 1.6732632423543772848170429916717 and 1.0507009873554804934193349852946 respectively.

# PyTorch implementation

import torch.nn.functional as F

def selu(x):

alpha = 1.6732632423543772848170429916717

scale = 1.0507009873554804934193349852946

return scale \* F.elu(x, alpha)

# Numpy implementation

import numpy as np

def selu(x):

alpha = 1.6732632423543772848170429916717

scale = 1.0507009873554804934193349852946

return scale \* ((x > 0)\*x + (x <= 0) \* (alpha \* np.exp(x) - alpha))

SNLI dataset

[SNLI dataset](https://nlp.stanford.edu/projects/snli/) is a collection of 570k english sentence pairs. The task is to classify each pair as either:

entailment - “A soccer game with multiple males playing.” and “Some men are playing a sport.”

contradiction - “A man inspects the uniform of a figure in some East Asian country.” and “The man is sleeping.”

neutral - “A smiling costumed woman is holding an umbrella.” and “A happy woman in a fairy costume holds an umbrella.”

Model Architecture

I am using a simple Bag of words model written in keras. The following python snippet describes the major components. This code is taken from Stephen Merity’s repo [here](https://github.com/Smerity/keras_snli).

# Embedding layer

embed = Embedding(VOCAB, EMBED\_HIDDEN\_SIZE, weights=[embedding\_matrix], input\_length=MAX\_LEN, trainable=False)

# A dense layer applied over each sequence point

translate = TimeDistributed(Dense(SENT\_HIDDEN\_SIZE, activation=ACTIVATION))

# A layer to sum up the sequence of words

rnn = keras.layers.core.Lambda(lambda x: K.sum(x, axis=1), output\_shape=(SENT\_HIDDEN\_SIZE, ))

# 2 pairs of input sentences

premise = Input(shape=(MAX\_LEN, ), dtype='int32')

hypothesis = Input(shape=(MAX\_LEN, ), dtype='int32')

# Get the word embeddings for each of these 2 pairs

prem = embed(premise)

hypo = embed(hypothesis)

# Apply the Dense layer

prem = translate(prem)

hypo = translate(hypo)

# Sum up the sequence

prem = rnn(prem)

hypo = rnn(hypo)

prem = BatchNormalization()(prem)

hypo = BatchNormalization()(hypo)

# Combined the 2 sentences

joint = concatenate([prem, hypo])

joint = Dropout(DP)(joint)

# Add Few dense layers in the end

for i in range(3):

joint = Dense(2 \* SENT\_HIDDEN\_SIZE, activation=ACTIVATION, kernel\_regularizer=l2(L2)(joint)

joint = Dropout(DP)(joint)

joint = BatchNormalization()(joint)

SELU vs RELU results

Code for this excercise is available in [this repo](https://github.com/hardikp/selu_snli).

RELU is clearly converging much faster than SELU. My first was to remove the BatchNormalization and do the same comparison. The following graph shows the comparison after removing the BatchNorm components.

Still, RELU seems to be doing a much better job than SELU for the default configuration.

This behavior remains more or less the same after iterating through hyperparameters. The following graph is for one of the hyperparameter configurations.

(Edit: I incorporated the suggestion from Dan Ofer below and included the graph with AlphaDropout.)

To be fair, it is still possible that SELU is better in some configurations. Some of the possible reasons are listed below. However, it is clear to me that simply replacing RELU with SELU isn’t going to improve your existing models.

SELU authors recommend a specific initialization scheme for it to be effective.

Additionally, SELU is a bit more computationally expensive than RELU. On a g2.2xlarge EC2 instance, RELU model took about 49 seconds to complete an epoch, while SELU model took 65 seconds to do the same (33% more).

4.In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?

In this blog, I will try to compare and analysis Sigmoid( logistic) activation function with others like Tanh, ReLU, Leaky ReLU, Softmax activation function. In my previous blog, I described on how to work sigmoid function in logistic Regression algorithm. There , I described with mathematical term and python implementation code. If you want to read, you may visit the [link](https://medium.com/@cmukesh8688/logistic-regression-sigmoid-function-and-threshold-b37b82a4cd79). So now i want to analysis more sigmoid function with another activation functions. Let’s move . These all are activation function used generally in Neural Network algorithm and deep learning. Here I don’t go in depth detail about Neural Network . We try to focus only activation functions. We will discuss Neural Network on another blog.

Basic idea of a Neural Network works-

There are many algorithms in the market to solve classification problem . Neural Network is one of them which is very famous for predicting accurate data. However, it takes a lot of computational time.It is inspired by the way biological neural systems process data. It contains layers of interconnected nodes or neurons arranged in interconnected layers.

The information moves from the input layer to the hidden layers. In a simple case of each layer, we just multiply the inputs by the weights, add a bias and apply an activation function to the result and pass the output to the next layer. We keep repeating this process until we reach the last layer.

Activation Function:

Activation functions are generally two types, These are

Linear or Identity Activation Function

Non-Linear Activation Function.

Generally, neural networks use non-linear activation functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions.They allow back-propagation because they have a derivative function which is related to the inputs.

Non-linear Activation Functions:

Above listed all activation functions are belong to non-linear activation functions. And we will discuss below more in details.

Sigmoid Activation Function:

Sigmoid Activation function is very simple which takes a real value as input and gives probability that ‘s always between 0 or 1. It looks like ‘S’ shape.

Sigmoid function and it’s derivative

It’s non-linear, continuously differentiable, monotonic, and has a fixed output range. Main advantage is simple and good for classifier. But Big disadvantage of the function is that it It gives rise to a problem of “vanishing gradients” because Its output isn’t zero centered. It makes the gradient updates go too far in different directions. 0 < output < 1, and it makes optimization harder. That takes very high computational time in hidden layer of neural network

# sigmoid function  
def sigmoid(z):  
 return 1.0 / (1 + np.exp(-z))# Derivative of sigmoid function  
def sigmoid\_prime(z):  
 return sigmoid(z) \* (1-sigmoid(z))

2. Tanh or Hyperbolic tangent:

Tanh help to solve non zero centered problem of sigmoid function. Tanh squashes a real-valued number to the range [-1, 1]. It’s non-linear too.

Derivative function give us almost same as sigmoid’s derivative function.

It solve sigmoid’s drawback but it still can’t remove the vanishing gradient problem completely.

When we compare tanh activation function with sighmoid , this picture give you clear idea.

# tanh activation function  
def tanh(z):  
 return (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z))# Derivative of Tanh Activation Function  
def tanh\_prime(z):  
 return 1 - np.power(tanh(z), 2)

3. ReLU (Rectified Linear Unit):

This is most popular activation function which is used in hidden layer of NN.The formula is deceptively simple: 𝑚𝑎𝑥(0,𝑧)max(0,z). Despite its name and appearance, it’s not linear and provides the same benefits as Sigmoid but with better performance.

ReLU Activation Function and It’s derivative

It’s main advantage is that it avoids and rectifies vanishing gradient problem and less computationally expensive than tanh and sigmoid. But it has also some draw back . Sometime some gradients can be fragile during training and can die. That leads to dead neurons.In another words, for activations in the region (x<0) of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input ( simply because gradient is 0, nothing changes ). So We should be very carefully to choose activation function , and activation function should be as per business requirement.

When we compare with sigmoid activation function, It’s look like

# ReLU activation function  
def relu(z):  
 return max(0, z)# Derivative of ReLU Activation Function  
def relu\_prime(z):  
 return 1 if z > 0 else 0

4. Leaky ReLU

It prevents dying ReLU problem.T his variation of ReLU has a small positive slope in the negative area, so it does enable back-propagation, even for negative input values

Leaky ReLU does not provide consistent predictions for negative input values. During the front propagation if the learning rate is set very high it will overshoot killing the neuron.

The idea of leaky ReLU can be extended even further. Instead of multiplying x with a constant term we can multiply it with a hyper-parameter which seems to work better the leaky ReLU. This extension to leaky ReLU is known as Parametric ReLU.

While we compare Leaky-ReLU with ReLU, then It shows clear concept of difference between them.

# Leaky\_ReLU activation function  
def leakyrelu(z, alpha):  
 return max(alpha \* z, z)# Derivative of leaky\_ReLU Activation Function  
def leakyrelu\_prime(z, alpha):  
 return 1 if z > 0 else alpha

5. Softmax

Generally, we use the function at last layer of neural network which calculates the probabilities distribution of the event over ’n’ different events. The main advantage of the function is able to handle multiple classes.

when we compare the sigmoid and softmax activation functions , they produce different results.

Sigmoid input values: -0.5, 1.2, -0.1, 2.4

Sigmoid output values: 0.37, 0.77, 0.48, 0.91

SoftMax input values: -0.5, 1.2, -0.1, 2.4

SoftMaxoutput values: 0.04, 0.21, 0.05, 0.70

Sigmoid’s probabilities produced by a Sigmoid are independent. Furthermore, they are not constrained to sum to one: 0.37 + 0.77 + 0.48 + 0.91 = 2.53. The reason for this is because the Sigmoid looks at each raw output value separately. Whereas Softmax’s the outputs are interrelated. The Softmax probabilities will always sum to one by design: 0.04 + 0.21 + 0.05 + 0.70 = 1.00. In this case, if we want to increase the likelihood of one class, the other has to decrease by an equal amount.

5.What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?

If you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer, then the algorithm will likely pick up a lot of speed, hopefully moving roughly toward the global minimum, but its momentum will carry it right past the minimum.

Hyper Parameter—Momentum

When we use the SGD (stochastic mini-batch gradient descent, commonly known as SGD in deep learning) to train parameters, sometimes it decreases very slowly and may fall into the local minimum value, or even zero, as shown in Fig 1 (the picture is from li hongyi, 《one day understanding deep learning》). But it is not always what you want.

There are three places where this pseudo-optimal solution occurs:

Plateau;

Saddle point;

Local minima.

Momentum is introduced to speed up the learning process, especially for the gradient with high curvature, small but consistent, which can accelerate the learning process. The main idea of momentum is to accumulate the moving average of previous gradients decaying exponentially.

It doesn’t necessarily mean that it’s going to be optimal using momentum. To illustrate these points in Fig 2 (the picture is from li hongyi, 《one day understanding deep learning》), in which red represents the direction of gradient descent, dotted green represents the direction of momentum, and blue represents the direction of actual movement :

For the first point. The direction of the gradient descent is to the right, but since we set it, there is no momentum at the beginning, so the actual direction of movement is the direction of the gradient descent.

For the second point. The gradient is going down to the right, but now the ball has a momentum that is moving to the right, which causes the ball to continue moving to the right.

For the third point. Since it is a local minimum, the gradient value is 0. If it’s normal gradient descent, it’s going to get stuck here. But we also have an impulse to the right, so using momentum, it’s actually moving to the right.

For the fourth point. Now our gradient descent is going to the left, and we can assume that if the impulse here is bigger than the gradient is going to be. At this point, the ball will continue in the direction of momentum, and it can even get out of the mountain, and get out of the local minima.

To better understanding momentumn, we create a random signal, and plot the figure when betas are 0.5,0.7,0.9,0.99.

>>> x = torch.linspace(-4, 4, 200)  
>>> y = torch.randn(200) + 0.3  
>>> betas = [0.5, 0.7, 0.9, 0.99]

The regular momentum function is shown below:

res = beta\*avg+yi

Including that res represents the result, beta is the value of momentumn, avg is 0.3 in the axample, and yi is the ith number.

The result is shown in Fig 3. When the beta is small, it gets a little bumpy. When beta is 0.99, the answer is totally wrong. The reason is that if the momentum is high, the basics you’re away from where you need to be in weight space. It’s literally biased to end up being a higher gradient than the actual gradient.

So we can fix that with modifying the momentum function:

res = beta\*v1 + (1-beta)\*v2.

It’s quite an exponentially weighted moving average as we know. It dampens the thing that we’re adding in. We can see the result is shown in Fig 4.

We can see it gets to a zero-constant when the data is purely random.

But in fact, we always deal with the signal that is specific, does that work well? To find out it, we create a new function here. The momentum is res = beta\*v1 + (1-beta)\*v2.

>>> y = 1 — (x/3) \*\* 2 + torch.randn(200) \* 0.1  
>>> y[0]=0.5

At the start, we add an outlier. The result is shown in Fig 5, it shows we’ve got in trouble. When beta is 0.99, in the momentum function, item number two is 0.99 times item number one plus 0.01 times item number two. The number one is massively biasing this data, it takes a very long time to close to the number.

There is also a problem that we’re always running a bit behind where we should be because we always only taking point one times a new thing.

To get a better result, we can use debiasing. Debiasing is to correct the wrong information we may have in the very first batch.

And the function is:

res = avg/(1-beta\*\*(i+1))

6.Name three ways you can produce a sparse model.

Features with sparse data are features that have mostly zero values. This is different from features with missing data. Examples of sparse features include vectors of one-hot-encoded words or counts of categorical data. On the other hand, features with dense data have predominantly non-zero values.

What is the Difference between Sparse Data and Missing Data?

When there is missing data, it means that many data points are unknown. On the other hand, if the data is sparse, all the data points are known, but most of them have zero value.

To illustrate this point, there are two types of features. The feature with sparse data has known values (= 0), but the feature with missing data has unknown values (= null). It is unknown what values should be in the null-valued rows.  
Table 1. Sample data with two types of features.

Why is Machine Learning Difficult with Sparse Features?

 Common problems with sparse features include:

If the model has many sparse features, it will increase the space and time complexity of models. Linear regression models will fit more coefficients, and tree-based models will have greater depth to account for all features.

Model algorithms and diagnostic measures might behave in unknown ways if the features have sparse data. [Kuss [2002]](https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.1421" \t "_blank) shows that goodness-of-fit tests are flawed when the data is sparse.

If there are too many features, models fit the noise in the training data. This is called overfitting. When models overfit, they are unable to generalize to newer data when they are put in production. This negatively impacts the predictive power of models.

Some models may underestimate the importance of sparse features and given preference to denser features even though the sparse features may have predictive power. Tree-based models are notorious for behaving like this. For example, random forests overpredict the importance of features that have more categories than those features that have fewer categories.

Methods for Dealing with Sparse Features

1. Removing features from the model

Sparse features can introduce noise, which the model picks up and increase the memory needs of the model. To remedy this, they can be dropped from the model. For example, rare words are removed from text mining models, or [features with low variance are removed](https://scikit-learn.org/stable/modules/feature_selection.html#removing-features-with-low-variance). However, sparse features that have important signals should not be removed in this process.

LASSO regularization can be used to decrease the number of features. Rule-based methods like setting a variance threshold for including features in the model might also be useful.

2. Make the features dense

Principal component analysis (PCA): PCA methods can be used to project the features into the directions of the principal components and select from the most important components.

[Feature hashing](https://en.wikipedia.org/wiki/Feature_hashing): In feature hashing, sparse features can be binned into the desired number of output features using a hash function. Care must be taken to choose a generous number of output features to prevent hash collisions.

3. Using models that are robust to sparse features

Some versions of machine learning models are robust towards sparse data and may be used instead of changing the dimensionality of the data. For example, the [entropy-weighted k-means algorithm](https://ieeexplore.ieee.org/abstract/document/4262534) is better suited to this problem than the regular k-means algorithm.

7.Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC Dropout?

Yes, dropout does slow down training, in general roughly by a factor of two. However, it has no impact on inference speed since it is only turned on during training. MC Dropout is exactly like dropout during training, but it is still active during inference, so each inference is slowed down slightly.

Dropout in Neural Networks

Dropout layers have been the go-to method to reduce the overfitting of neural networks. It is the underworld king of regularisation in the modern era of deep learning.

In this era of deep learning, almost every data scientist must have used the dropout layer at some moment in their career of building neural networks. But, why dropout is so common? How does the dropout layer work internally? What is the problem that it solves? Is there any alternative to dropout?

If you have similar questions regarding dropout layers, then you are in the correct place. In this blog, you will discover the intricacies behind the famous dropout layers. After completing this blog, you would be comfortable answering different queries related to dropout and if you are more of an innovative person, you might come up with a more advanced version of dropout layers.

Let’s start… :)

OVERVIEW

This blog is divided into the following sections:

Introduction: The problem it tries to solve

What is a dropout?

How does it solve the problem?

Dropout Implementation

Dropout during Inference

How it was conceived

Tensorflow implementation

Conclusion

INTRODUCTION

So before diving deep into its world, let’s address the first question. What is the problem that we are trying to solve?

The deep neural networks have different architectures, sometimes shallow, sometimes very deep trying to generalise on the given dataset. But, in this pursuit of trying too hard to learn different features from the dataset, they sometimes learn the statistical noise in the dataset. This definitely improves the model performance on the training dataset but fails massively on new data points (test dataset). This is the problem of overfitting. To tackle this problem we have various regularisation techniques that penalise the weights of the network but this wasn’t enough.

The best way to reduce overfitting or the best way to regularise a fixed-size model is to get the average predictions from all possible settings of the parameters and aggregate the final output. But, this becomes too computationally expensive and isn’t feasible for a real-time inference/prediction.

The other way is inspired by the ensemble techniques (such as AdaBoost, XGBoost, and Random Forest) where we use multiple neural networks of different architectures. But this requires multiple models to be trained and stored, which over time becomes a huge challenge as the networks grow deeper.

So, we have a great solution known as Dropout Layers.

What is a Dropout?

The term “dropout” refers to dropping out the nodes (input and hidden layer) in a neural network (as seen in Figure 1). All the forward and backwards connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network. The nodes are dropped by a dropout probability of p.

Let’s try to understand with a given input x: {1, 2, 3, 4, 5} to the fully connected layer. We have a dropout layer with probability p = 0.2 (or keep probability = 0.8). During the forward propagation (training) from the input x, 20% of the nodes would be dropped, i.e. the x could become {1, 0, 3, 4, 5} or {1, 2, 0, 4, 5} and so on. Similarly, it applied to the hidden layers.

For instance, if the hidden layers have 1000 neurons (nodes) and a dropout is applied with drop probability = 0.5, then 500 neurons would be randomly dropped in every iteration (batch).

Generally, for the input layers, the keep probability, i.e. 1- drop probability, is closer to 1, 0.8 being the best as suggested by the authors. For the hidden layers, the greater the drop probability more sparse the model, where 0.5 is the most optimised keep probability, that states dropping 50% of the nodes.

So how does dropout solves the problem of overfitting?

How does it solve the Overfitting problem?

In the overfitting problem, the model learns the statistical noise. To be precise, the main motive of training is to decrease the loss function, given all the units (neurons). So in overfitting, a unit may change in a way that fixes up the mistakes of the other units. This leads to complex co-adaptations, which in turn leads to the overfitting problem because this complex co-adaptation fails to generalise on the unseen dataset.

Now, if we use dropout, it prevents these units to fix up the mistake of other units, thus preventing co-adaptation, as in every iteration the presence of a unit is highly unreliable. So by randomly dropping a few units (nodes), it forces the layers to take more or less responsibility for the input by taking a probabilistic approach.

This ensures that the model is getting generalised and hence reducing the overfitting problem.

From figure 2, we can easily make out that the hidden layer with dropout is learning more of the generalised features than the co-adaptations in the layer without dropout. It is quite apparent, that dropout breaks such inter-unit relations and focuses more on generalisation.

Dropout Implementation

Enough of the talking! Let’s head to the mathematical explanation of the dropout.

(a) A unit (neuron) during training is present with a probability p and is connected to the next layer with weights ‘w’ ; (b) A unit during inference/prediction is always present and is connected to the next layer with weights, ‘pw’ (Image by [Nitish](https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf))

In the original implementation of the dropout layer, during training, a unit (node/neuron) in a layer is selected with a keep probability (1-drop probability). This creates a thinner architecture in the given training batch, and every time this architecture is different.

In the standard neural network, during the forward propagation we have the following equations:

where:  
z: denote the vector of output from layer (l + 1) before activation  
y: denote the vector of outputs from layer l  
w: weight of the layer l  
b: bias of the layer l

Further, with the activation function, z is transformed into the output for layer (l+1).

Now, if we have a dropout, the forward propagation equations change in the following way:

So before we calculate z, the input to the layer is sampled and multiplied element-wise with the independent Bernoulli variables. r denotes the Bernoulli random variables each of which has a probability p of being 1. Basically, r acts as a mask to the input variable, which ensures only a few units are kept according to the keep probability of a dropout. This ensures that we have thinned outputs “y(bar)”, which is given as an input to the layer during feed-forward propagation.

Dropout during Inference

Now, we know the dropout works mathematically but what happens during the inference/prediction? Do we use the network with dropout or do we remove the dropout during inference?

This is one of the most important concepts of dropout which very few data scientists are aware of.

According to the original implementation (Figure 3b) during the inference, we do not use a dropout layer. This means that all the units are considered during the prediction step. But, because of taking all the units/neurons from a layer, the final weights will be larger than expected and to deal with this problem, weights are first scaled by the chosen dropout rate. With this, the network would be able to make accurate predictions.

To be more precise, if a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p during the prediction stage.

8.Practice training a deep neural network on the CIFAR10 image dataset:

* 1. Build a DNN with 20 hidden layers of 100 neurons each (that’s too many, but it’s the point of this exercise). Use He initialization and the ELU activation function.
  2. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with keras.datasets.cifar10.load\_​data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you’ll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters.
  3. Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed?
  4. Try replacing Batch Normalization with SELU, and make the necessary adjustements to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
  5. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.

Beginner Friendly Image Classification using PyTorch and the CIFAR-10 dataset

Photo by [Joshua Sortino](https://unsplash.com/@sortino?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Neural Networks: How do they work?

Neural networks are versatile models that can learn just about any complex pattern. These powerful models are the core of deep learning consisting of multi-layer perceptrons, convolutional networks, sequence models and many more. In this brief project, I will explore the [CIFAR-10 dataset](https://www.cs.toronto.edu/~kriz/cifar.html) and implement a simple neural network (multi-layer perceptron).

The concept of a neural network is actually quite simple. Similar to how neurons fire or activate in the human brain, the neurons within a layer in a neural network are activated through an activation function. This process returns output that will be passed on to the next layer of the neural network and the cycle is repeated until the end of the neural network. This process is known as the forward pass where your data is fed forward through the network after applying weights and an activation function. Depending on whether you are solving a regression or classification problem, the final output layer of your neural network will output one node for regression problems and multiple nodes for classification problems. In this project, we will be classifying images so the output from the neural network will have one node per class which will go through the softmax function to obtain the final prediction.

The essence of neural networks is not what happens during the forward pass but rather what happens after the forward pass. The final output values from a forward pass are used to update the weights of the neural network by going backwards through the network. This is known as backpropagation which is an extremely powerful algorithm that updates the weights iteratively from the last layer to the first. It is an extremely efficient way to train neural networks when using a stochastic gradient descent optimizer.

Preparation for model training

As stated from the CIFAR-10 information page, this dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. Since we are working with coloured images, our data will consist of numeric values that will be split based on the RGB scale.

Sample Image from the CIFAR-10 dataset

For this project, we will use 10% of the dataset as the validation set and 90% as the training set. The loss function will be cross-entropy loss since this is a classification problem. The optimizer will be stochastic gradient descent and the batch size for gradient descent will be 128. Stochastic gradient descent is an approximation of gradient descent. The gradient of the loss function is applied to a batch of all the training points instead of the whole set, which is much faster to compute. This stochastic batch sampling of training samples introduces a lot of noise, which is actually helpful in preventing the algorithm from getting stuck in narrow local minima.

Base Model and Training on GPU

First, we create the base model for our neural network where we will define functions for the training process and validation process.

Image Classification Base Model

Then we will define the evaluate function to return the progress of our model after each epoch and the fit function which will be used to update the weights for each epoch.

Model evaluation function and fit function

Since we are using PyTorch, we have the option to use the GPU for training and evaluating our model. GPUs are much more efficient for updating and calculating weights because they are optimized for matrix calculations as opposed to the CPU. Therefore, we will move our data to the GPU if it is available.

Moving data to GPU

Now we are ready to define our neural network. For this example, I will be using 4 hidden layers for my neural network with input nodes of 1536, 768, 384, and 128 respectively for each layer. The forward pass function will use the rectified linear unit activation function (ReLu) which is a transition function that applies max(x, 0) to an input x. This transition function is popular for neural networks since it will not activate all nodes within a layer making the model non-linear.

Creating hidden layers and forward pass function

Finally, we are ready to train and evaluate the model. After initializing random weights, we will iterate through the neural network and backpropogate depending on our learning rate (step sizes) and epochs. The goal here is to obtain weights that ultimately minimize the loss. We will do this by taking big steps through gradient descent until we converge closer to the minimum. Once we converge closer to the minimum, we will take smaller step sizes until we reach the lowest possible minimum.

15 epochs at learning rate 0.1

5 epochs at learning rate 0.01

5 epochs at learning rate 0.001

5 epochs at learning rate 0.0001 (loss seems to be converging at this point)

We know that we’ve converged to the minimum because our accuracy does not improve any further and our loss does not decrease any further. This is the best model our neural network can come up with. This means that our model is correct a little more than half the time. The reason for this could be because our model looks at each pixel rather than the whole picture. When we look at images, it’s relatively hard to generalize what an image is if we looked at only one pixel.

Loss vs Epochs

Accuracy vs Epochs

Now that we know we’ve gotten the best model possible, let us test it against the testing set.

Evaluation with the testing set

An accuracy of 55%! This goes to show that it is crucial to pick the right number of epochs and learning rates in order to find the best model for your dataset but in this case, it would not be practical to use this model to classify anything since the accuracy is quite low. As stated earlier, this is most likely due to the fact that the model is looking at each pixel one by one. This is where convolution networks shine. These networks are deep neural networks that make use of convolution layers which use convolutional filters to process and produce images. Sort of how humans can get a better idea of what an image is when we look at the full picture or parts of a picture, convolution networks are able to look at a portion of a picture, allowing it to retain more information about an image rather than looking at a pixel.