sas

Deep Learning for Computer Vision with SAS®

An Introduction

Robert Blanchard

The correct bibliographic citation for this manual is as follows: Blanchard, Robert 2020. *Deep Learning for Computer Vision with SAS®: An Introduction*. Cary, NC: SAS Institute Inc.

Deep Learning for Computer Vision with SAS®: An Introduction

ISBN 978-1-64295-972-7 (Hardcover)
ISBN 978-1-64295-915-4 (Paperback)
ISBN 978-1-64295-916-1 (PDF)
ISBN 978-1-64295-917-8 (EPUB)
ISBN 978-1-64295-918-5 (Kindle)

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

June 2020

Contents

About This Book

What Does This Book Cover?

Deep learning is an area of machine learning that has become ubiquitous with artificial intelligence. The complex, brain-like structure of deep learning models is used to find intricate patterns in large volumes of data. These models have heavily improved the performance of general supervised models, time series, speech recognition, object detection and classification, and sentiment analysis.

SAS has a rich set of established and unique capabilities with regard to deep learning. This book introduces the basics of deep learning with a focus on computer vision. The book details and demonstrates how to build computer vision models using SAS software. Both the “art” and science behind model building is covered.

Is This Book for You?

The general audience for this book should be either SAS or Python programmers with knowledge of traditional machine learning methods.

What Should You Know about the Examples?

This book includes tutorials for you to follow to gain hands-on experience with SAS.

Software Used to Develop the Book's Content

To follow along with the demos in this book, you will need the following software:

- SAS Viya (VDMML)
- SAS Studio
- Python

Example Code and Data

You can access the example code and data for this book by linking to its author page at https://support.sas.com/blanchard or on GitHub at https://github.com/sassoftware.

We Want to Hear from You

SAS Press books are written *by* SAS Users *for* SAS Users. We welcome your participation in their development and your feedback on SAS Press books that you are using. Please visit sas.com/books to do the following:

- Sign up to review a book
- Recommend a topic
- Request information on how to become a SAS Press author
- Provide feedback on a book

Do you have questions about a SAS Press book that you are reading? Contact the author through saspress@sas.com or https://support.sas.com/author_feedback.

SAS has many resources to help you find answers and expand your knowledge. If you need additional help, see our list of resources: sas.com/books.

Learn more about this author by visiting his author page https://support.sas.com/blanchard. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

About The Author

Robert Blanchard is a Senior Data Scientist at SAS where he builds end-to-end artificial intelligence applications. He also researches, consults, and teaches machine learning with an emphasis on deep learning and computer vision for SAS. Robert has authored several professional courses on topics including neural networks, deep learning, and optimization modeling. Before joining SAS, Robert worked under the Senior Vice Provost at North Carolina State University, where he built models pertaining to student success, faculty development, and resource management. While working at North Carolina State University, Robert also started a private analytics company that focused on predicting future home sales. Prior to working in academia, Robert was a member of the research and development group on the Workforce Optimization team at Travelers Insurance. His models at Travelers focused on forecasting and optimizing resources. Robert graduated with a master's degree in Business Analytics and Project Management from the University of Connecticut and a master's degree in Applied and Resource Economics from East Carolina University.

Learn more about this author by visiting his author page https://support.sas.com/blanchard. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

Chapter 1: Introduction to Deep Learning

Introduction to Neural Networks

Artificial neural networks mimic key aspects of the brain, in particular, the brain's ability to learn from experience. In order to understand artificial neural networks, we first must understand some key concepts of biological neural networks, in other words, our own biological brains.

A biological brain has many features that would be desirable in artificial systems, such as the ability to learn or adapt easily to new environments. For example, imagine you arrive at a city in a country that you have never visited. You don't know the culture or the language. Given enough time, you will learn the culture and familiarize yourself with the language. You will know the location of streets, restaurants, and museums.

The brain is also highly parallel and therefore very fast. It is not equivalent to one processor, but instead it is equivalent to a multitude of millions of processors, all running in parallel. Biological brains can also deal with information that is fuzzy, probabilistic, noisy, or inconsistent, all while being robust, fault tolerant, and relatively small. Although inspired by cognitive science (in particular, neurophysiology), neural networks largely draw their methods from statistical physics (Hertz et al. 1991). There are dozens, if not hundreds, of neural network algorithms.

Biological Neurons

In order to imitate neurons in artificial systems, first their mechanisms needed to be understood. There is still much to be learned, but the key functional aspects of neurons, and even small systems (networks) of neurons, are now known.

Neurons are the fundamental units of cognition, and they are responsible for sending information from the brain to the rest of the body. Neurons have three parts: a cell body, dendrites, and axons. Inputs arrive in the dendrites (short branched structures) and are transmitted to the next neuron in the chain via the axons (a long, thin fiber). Neurons do not actually touch each other but communicate across the gap (called a synaptic gap) using neurotransmitters. These chemicals either excite the receiving neuron, making it more likely to "fire," or they inhibit the neuron, making it less likely to become active. The amount of neurotransmitter released across the gap determines the relative strength of each dendrite's connection to the receiving neuron. In essence, each synapse "weights" the relative strength of its arriving input. The synaptically weighted inputs are summed. If the sum exceeds an adaptable threshold (or bias) value, the neuron sends a pulse down its axon to the other neurons in the network to which it connects.

A key discovery of modern neurophysiology is that synaptic connections are adaptable; they change with experience. The more active the synapse, the stronger the connection becomes. Conversely, synapses with little or no activity fade and, eventually, die off (atrophy). This is thought to be the basis of learning. For example, a study from the University of Wisconsin in 2015 showed that people could begin to "see" with their tongue. Attached to the electric grid was a camera that was fastened to the subject's forehead. The subject was blindfolded. However, within 30 minutes, as their neurons adapted, subjects began to "see" with their tongue. Amazing!

Although there are branches of neural network research that attempt to mimic the underlying biological processes in detail, most neural networks do not try to be biologically realistic.

Mathematical Neurons

In a seminal paper with the rather understated title "A logical calculus of the ideas immanent in nervous activity," McCulloch and Pitts (1943) gave birth to the field of artificial neural networks. The fundamental element of a McCulloch-Pitts network is called, unsurprisingly, a McCulloch-Pitts neuron. As in real neurons, each input (x_i) is first weighted (w_i) and then summed. To mimic a neuron's threshold functionality, a bias value (w_0) is added to the weighted sum, predisposing the neuron to either a positive or negative output value. The result is known as the neuron's net input:

$$net = w_0 + \sum_{i=1}^{k} w_i x_i$$

Notice that this is the classic linear regression equation, where the bias term is the y-intercept and the weight associated with each input is the input's slope parameter.

The original McCulloch-Pitts neuron's final output was determined by passing its net input value through a step function (a function that converts a continuous value into a binary output 0 or 1, or a bipolar output -1 or 1), turning each neuron into a linear classifier/discriminator. Modern neurons replace the discontinuous step function used in the McCulloch-Pitts neuron with a continuous function. The continuous nature permits the use of derivatives to explore the parameter space.

$$H = f\left(w_0 + \sum_{i=1}^{d} w_i x_i \right)$$

The mathematical neuron is considered the cornerstone of a neural network. There are three layers in the basic multilayer perceptron (MLP) neural network:

1. An *input layer* containing a neuron/unit for each input variable. The input layer neurons have no adjustable parameters (weights). They simply pass the positive or negative input to the next layer.
2. A *hidden layer* with hidden units (mathematical neurons) that perform a nonlinear transformation of the weighted and summed input activations.
3. An *output layer* that shapes and combines the nonlinear hidden layer activation values.

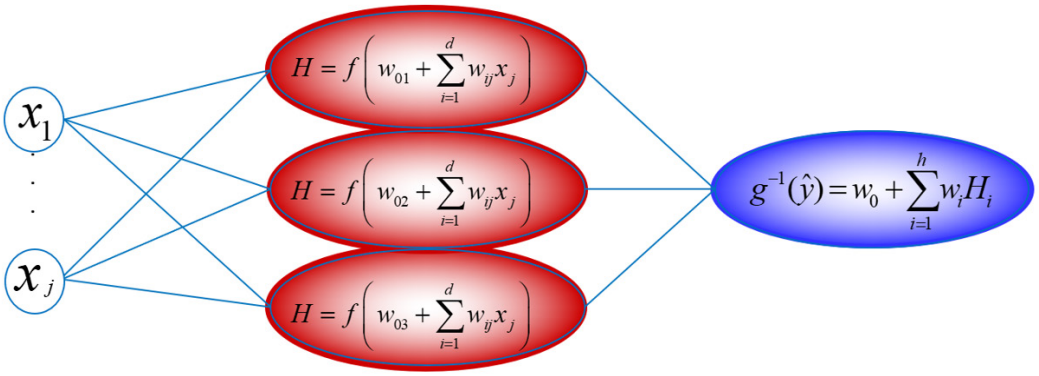
A single hidden-layer multilayer perceptron constructs a limited extent region, or *bump*, of large values surrounded by smaller values (Principe et al. 2000). The intersection of the hyper-planes created by a hidden layer consisting of three hidden units, for example, forms a triangle-shaped bump.

The hidden and output layers ***must not*** be connected by a strictly linear function in order to act as separate layers. Otherwise, the multilayer perceptron collapses into a linear perceptron. More formally, if matrix **A** is the set of weights that transforms input matrix **X** into the hidden layer output values, and matrix **B** is the set of weights that transforms the hidden unit output into the final estimates **Y**, then the linearly connected multilayer network can be represented as **Y**=**B**[**A**(**X**)]. However, if a single-layer weight matrix **C**=**BA** is created, exactly the same output can be obtained from the single-layer network—that is, **Y**=**C**(**X**).

In a two-layer perceptron with *k* inputs, h_1 hidden units in the first hidden layer, and h_2 hidden units in the second hidden layer, the number of parameters to be learned is $h_1(k+1) + h_2(h_1+1) = h_2 = 1$.

The number 1 represents the biased weight W_0 in the combination function of each neuron.

Figure 1.1: Multilayer Perceptron



Note: The "number of parameters" equations in this book assume that the inputs are interval or ratio level. Each nominal or ordinal input increases *k* by the number of classes in the variable, minus 1.

Deep Learning

The term *deep learning* refers to the numerous hidden layers used in a neural network. However*, the true essence of deep learning is the methods that enable the increased extraction of information* derived from a neural network with more than one hidden layer. Adding more hidden layers to a neural network provides little benefit without deep learning methods that underpin the efficient extraction of information. For example, SAS software has had the capability to build neural networks with many hidden layers using the NEURAL procedure for several decades. However, a case can be made to suggest that SAS has not had deep learning because the key elements that enable learning to persist in the presence of many hidden layers had not been discovered. These elements include the use of the following:

- activation functions that are more resistant to saturation than conventional activation functions
- fast moving gradient-based optimizations such as Stochastic Gradient Descent and ADAM
- weight initializations that consider the amount of incoming information
- new regularization techniques such as dropout and batch normalization
- innovations in distributed computing.

The elements outlined above are included in today's SAS software and are described below. Needless to say, deep learning has shown impressive promise in solving problems that were previously considered infeasible to solve.

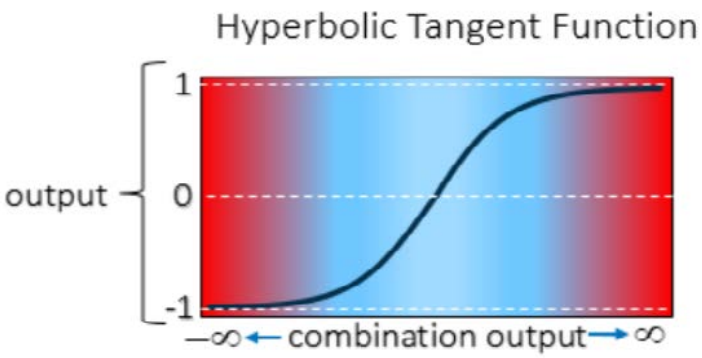
The process of deep learning is to formulate an outcome from engineering new glimpses of the input space, and then reengineering these engineered projections with the next hidden layer. This process is repeated for each hidden layer until the output layers are reached. The output layers reconcile the final layer of incoming hidden unit information to produce a set of outputs. The classic example of this process is facial recognition. The first hidden layer captures shades of the image. The next hidden layer combines the shades to formulate edges. The next hidden layer combines these edges to create projections of ears, mouths, noses, and other distinct aspects that define a human face. The next layer combines these distinct formulations to create a projection of a more complete human face. And so on. A brief comparison of traditional neural networks and deep learning is shown in Table 1.1.

Table 1.1: Traditional Neural Networks versus Deep Learning

Aspect	Traditional	Deep Learning
Hidden activation function(s)	Hyperbolic Tangent (tanh)	Rectified Linear (ReLU) and other variants
Gradient-based learning	Batch GD and BFGS	Stochastic GD, Adam, and LBFGS
Weight initialization	Constant Variance	Normalized Variance
Regularization	Early Stopping, L1, and L2	Early Stopping, L1, L2, Dropout, and Batch Normalization
Processor	CPU	CPU or GPU

Deep learning incorporates activation functions that are more resistant to neuron saturation than conventional activation functions. One of the classic characteristics of traditional neural networks was the infamous use of sigmoidal transformations in hidden units. Sigmoidal transformations are problematic for gradient-based learning because the sigmoid has two asymptotic regions that can saturate (that is, gradient of the output is near zero). The red or deeper shaded outer areas represent areas of saturation. See Figure 1.2.

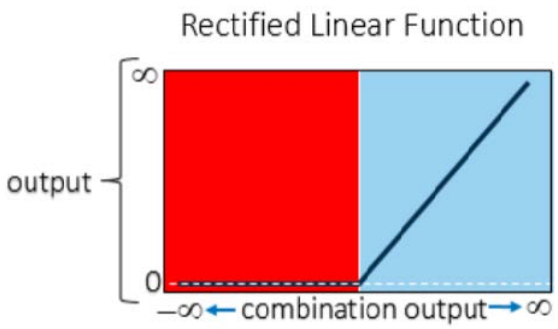
Figure 1.2: Hyperbolic Tangent Function



On the other hand, a linear transformation such as an identity poses little issue for gradient-based learning because the gradient is a constant. However, the use of linear transformations negates the benefits provided by nonlinear transformations (that is, approximate nonlinear relationships).

Rectified linear transformation (or ReLU) consists of piecewise linear transformations that, when combined, can approximate nonlinear functions. (See Figure 1.3.)

Figure 1.3: Rectified Linear Function



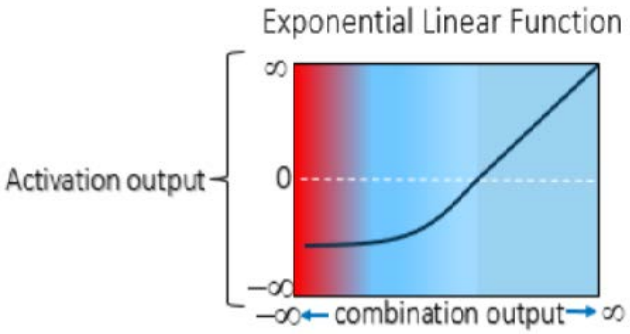
In the case of ReLU, the derivative for the ***active*** region output by the transformation is 1 and 0 for the ***inactive*** region. The inactive region of the ReLU transformation can be viewed as a weakness of the transformation because it inhibits the unit from contributing to gradient-based learning.

The saturation of ReLU could be somewhat mitigated by cleverly initializing the weights to avoid negative output values. For example, consider a business scenario of modeling image data. Each unstandardized input pixel value ranges between 0 and 255. In this case, the weights could be

initialized and constrained to be strictly positive to avoid negative output values, avoiding the non-active output region of the ReLU.

Other variants of the rectified linear transformation exist that permit learning to continue when the combination function resolves to a negative value. Most notable of these is the exponential linear activation transformation (ELU) as shown in Figure 1.4.

Figure 1.4: Exponential Linear Function



SAS researchers have observed better performance when ELU is used instead of ReLU in convolutional neural networks in some cases. SAS includes other, popular activation functions that are not shown here, such as softplus and leaky. Additionally, you can create your own activation functions in SAS using the SAS Function Compiler (or FCMP).

> **Note:** Convolutional neural networks (CNNs) are a class of artificial neural networks. CNNs are widely used in image recognition and classification. Like regular neural networks, a CNN consists of multiple layers and a number of neurons. CNNs are well suited for image data, but they can also be used for other problems such as natural language processing. CNNs are detailed in Chapter 2.

The error function defines a surface in the parameter space. If it is a linear model fit by least squares, the error surface is convex with a unique minimum. However, in a nonlinear model, this error surface is often a complex landscape consisting of numerous deep valleys, steep cliffs, and long-reaching plateaus.

To efficiently search this landscape for an error minimum, optimization must be used. The optimization methods use local features of the error surface to guide their descent. Specifically, the parameters associated with a given error minimum are located using the following procedure:

1. Initialize the weight vector to small random values, $\mathbf{w}^{(0)}$.
2. Use an optimization method to determine the update vector, $\delta^{(t)}$.

3. Add the update vector to the weight values from the previous iteration to generate new estimates:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \boldsymbol{\delta}^{(t)}$$

4. If none of the specified convergence criteria have been achieved, then go back to step 2.

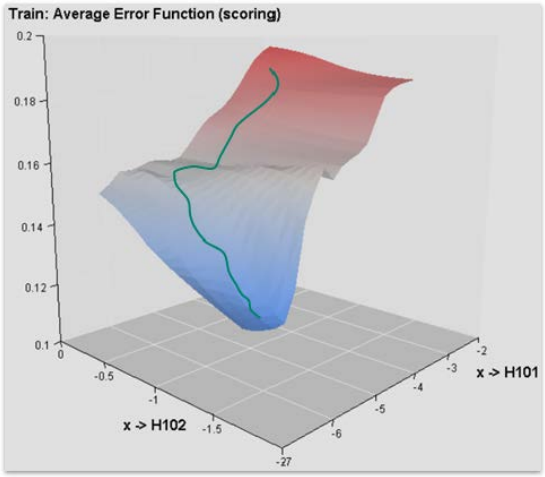
Here are the three conditions under which convergence is declared:

1. when the specified error function stops improving
2. if the gradient has no slope (implying that a minimum has been reached)
3. if the magnitude of the parameters stops changing substantially

Batch Gradient Descent

Re-invented several times, the back propagation (*backprop*) algorithm initially just used *gradient descent* to determine an appropriate set of weights. The gradient, $\nabla \mathbf{g}^{(t)}$, is the vector of partial derivatives of the error function with respect to the weights. It points in the steepest direction uphill. (See Figure 1.5.)

Figure 1.5: Batch Gradient Descent



By negating the step size (that is, *learning rate*) parameter, η, a step is made in the direction that is locally steepest downhill:

$$\boldsymbol{\delta}^{(t)} = -\eta \nabla \mathbf{g}^{(t)}$$

The parameters associated with a given error minimum are located using the following procedure:

1. Initialize the weight vector to small random values, $\mathbf{w}^{(0)}$.
2. Use an optimization method to determine the update vector, $\delta^{(t)}$.
3. Add the update vector to the weight values from the previous iteration to generate new estimates:

$$\mathrm{w}^{(t+1)} = \mathrm{w}^{(t)} + \delta^{(t)}$$

5. If none of the specified convergence criteria has been achieved, then back go to step 2.

Unfortunately, as gradient descent approaches the desired weights, it exhibits numerous back-and-forth movements known as *hemstitching*. To control the training iterations wasted in this hemstitching, later versions of back propagation included a momentum term, yielding the modern update rule:

$$\boldsymbol{\delta}^{(t)} = -\eta \nabla \mathbf{g}^{(t)} + \alpha \boldsymbol{\delta}^{(t-1)}$$

The momentum term retains the last update vector, $\delta^{(t-1)}$, using this information to "dampen" potentially oscillating search paths. The cost is an extra learning rate parameter ($0 \le \alpha \le 1$) that must be set. This updated rule uses all the training observations (t) to calculate the exact gradient on each descent step. This results in a smooth progression to the gradient minima.

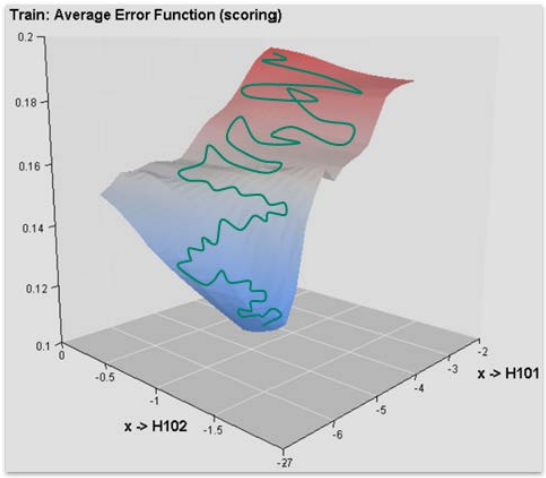
Stochastic Gradient Descent

In the batch variant of the gradient descent algorithm, generation of the weight update vector is determined by using all of the examples in the training set. That is, the exact gradient is calculated, ensuring a relatively smooth progression to the error minima.

However, when the training data set is large, computing the exact gradient is computationally expensive. The entire training data set must be assessed on each step down the gradient. Moreover, if the data are redundant, the error gradient on the second half of the data will be almost identical to the gradient on the first half. In this event, it would be a waste of time to compute the gradient on the whole data set. You would be better off computing the gradient on a subset of the weights, updating the weights, and then repeating on a new subset. In this case, each weight update is based on an approximation to the true gradient. But as long as it points in approximately the same direction as the exact gradient, the approximate gradient is a useful alternative to computing the exact gradient (Hinton 2007).

Taken to extremes, calculation of the approximate gradient can be based on a single training case. The weights are then updated, and the gradient is calculated on the next case. This is known as *stochastic gradient descent* (also known as *online learning*). (See Figure 1.6.)

Figure 1.6: Stochastic Gradient Descent



Stochastic gradient descent is very effective, particularly when combined with a momentum term, $\boldsymbol{\delta}^{(t-1)}$:

$$\boldsymbol{\delta}^{(t)} = -\eta \nabla \mathbf{g}^{(t)} + \alpha \boldsymbol{\delta}^{(t-1)}$$

Because stochastic gradient descent does not need to consider the entire training data set when calculating each descent step's gradient, it is usually faster than batch gradient descent. However, because each iteration is trying to better fit a single observation, some of the gradients might actually point away from the minima. This means that, although stochastic gradient descent generally moves the parameters in the direction of an error minima, it might not do so on each iteration. The result is a more circuitous path. In fact, stochastic gradient descent does not actually converge in the same sense as batch gradient descent does. Instead, it wanders around continuously in some region that is close to the minima (Ng, 2013).

Introduction to ADAM Optimization

The ADAM method applies adjustments to the learned gradients for each individual model parameter in an adaptive manner by approximating second-order information about the objective function based on previously observed mini-batch gradients. The "adaptive movement" nature of the algorithm's movement is where the name ADAM comes from (Kingma and Ba, 2014).

The ADAM method introduces two new hyperparameters to the mix, (β_1^t) and (β_2^t) where *t* represents the iteration count. A learning rate that controls the originating step size is also included. The adjustable beta terms are used to approximate a *signal-to-noise* ratio that is used to scale the step size. When the approximated single-to-noise ratio is large, the step size is closer to the originating step size (that of traditional stochastic gradient descent).

When the approximated single-to-noise ratio is small, the step size is near zero. This is a nice feature because a lower single-to-noise ratio is an indication of higher uncertainty. Thus, more cautious steps should be taken in the parameter space (Kingma and Ba 2014).

To use ADAM, specify 'ADAM' in the METHOD= suboption of the ALGORITHM= option in the OPTIMIZER parameter. The suboptions for β_1 and β_2, as well as the α and other options, also need to be specified. In the example code below, $\beta_1 = .9$, $\beta_2 = .999$ and $\alpha = .001$.

```
optimizer={algorithm={method='ADAM',
                      beta1=0.9,
                      beta2=0.999,
                      learningrate=.001,
                      lrpolicy='Step',
                      gamma=0.5},
                      minibatchsize=100,
                      maxepochs=200}
```

Note: The authors of ADAM recommend a β_1 value of .9, a β_2 value of .999, and an α (learning rate) of .001.

Weight Initialization

Deep learning uses different methods of weight initialization than traditional neural networks do. In neural networks, the hidden unit weights are randomly initialized to ensure that each hidden unit is approximating different areas of relationship between the inputs and the output. Otherwise, each hidden unit would be approximating the same relational variations if the weights across hidden units were identical, or even symmetric. The hidden unit weights are usually randomly initialized to some specified distribution, commonly Gaussian or Uniform.

Traditional neural networks use a standard variance for the randomly initialized hidden unit weights. This can become problematic when there is a large amount of incoming information (that is, a large number of incoming connections) because the variance of the hidden unit will likely increase as the amount of incoming connections increases. This means that the output of the combination function *could* be more extreme, resulting in a saturated hidden unit (Daniely et al. 2017).

Deep learning methods use a normalized initialization in which the variance of the hidden weights is a function of the amount of incoming information and outgoing information. SAS offers several methods for reducing the variance of the hidden weights. *Xavier* initialization is one of the most common weight initialization methods used in deep learning. The initialization method is random uniform with variance

$$w_{i,j} \sim U(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}})$$

where *m* is the number of input connections (fan-in) and *n* is the number of output connections (fan-out) (hidden units in current layer).

One potential flaw of the Xavier initialization is that the initialization method assumes a linear activation function, which is typically not the case in hidden units. *MSRA* was designed with the ReLU activation function in mind because MSRA operates under the assumption of a nonzero mean output by the activation, which is exhibited by ReLU (He et al. 2015). The MSRA initialization method is random Gaussian distribution with a standardization of

$$\sqrt{\frac{2}{avg(m+n)}}$$

SAS includes a second variant of the MSRA, called *MSRA2*. Similar to the MSRA initialization, the MSRA2 method is a random Gaussian distribution with a standardization of

$$\sqrt{\frac{2}{n}}$$

And it penalizes only for outgoing (fan-out) information.

> **Note:** Weight initializations have less impact over model performance if batch normalization is used because batch normalization standardizes information passed between hidden layers. Batch normalization is discussed later in this chapter.

Consider the following simple example where unit y is being derived from 25 randomly initialized weights. The variance of unit y is larger when the standard deviation is held constant at 1. This means that the values for y are more likely to venture into a saturation region when a nonlinear activation function is incorporated. On the other hand, Xavier's initialization penalizes the variance for the incoming and outgoing connections, constraining the value of y to less treacherous regions of the activation. See Figures 1.7 and 1.8, noting that these examples assume that there are 25 incoming and outgoing connections.

Figure 1.7: Constant Variance (Standard Deviation = 1)

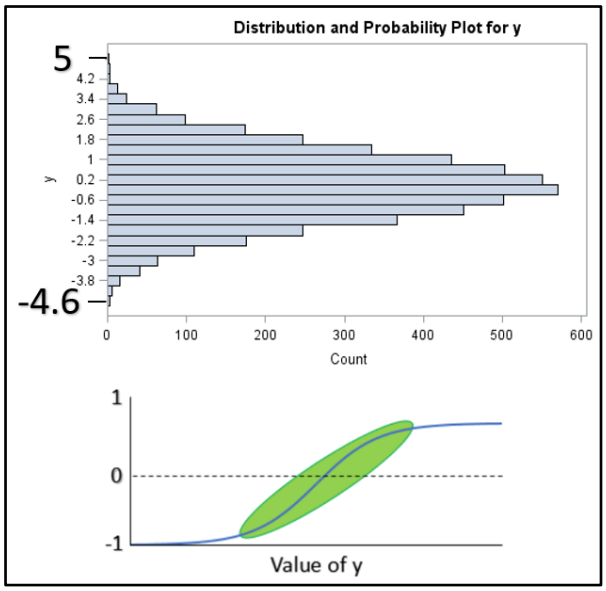
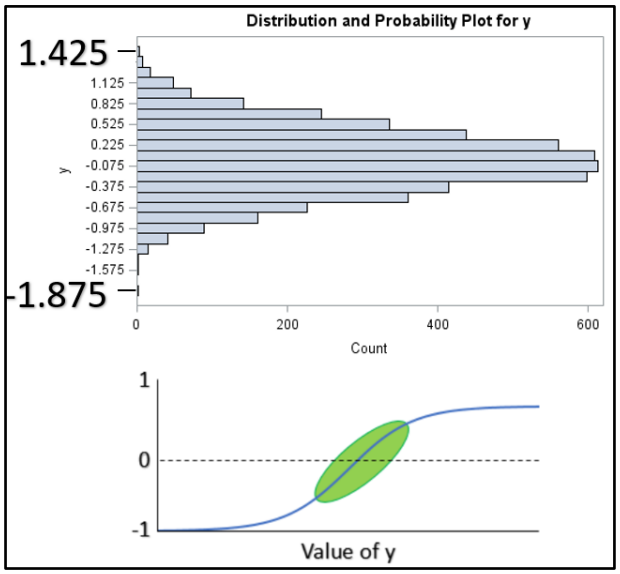


Figure 1.8: Constant Variance (Standard Deviation = $\sqrt{\frac{6}{25+25}} \approx .34$)



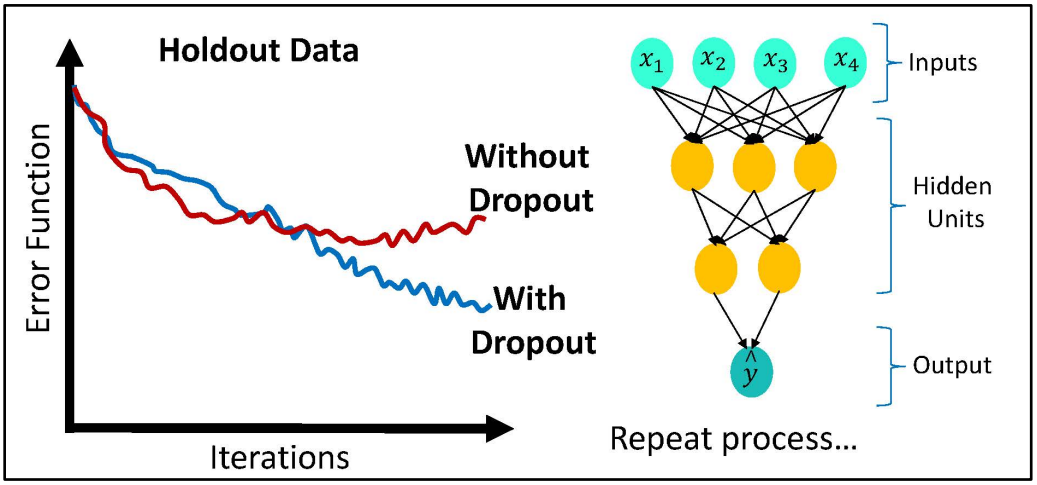
Regularization

Regularization is a process of introducing or removing information to stabilize an algorithm's understanding of data. Regularizations such as early stopping, L1, and L2 have been used extensively in neural networks for many years. These regularizations are still widely used in deep learning, as well. However, there have been advancements in the area of regularization that

work particularly well when combined with multi-hidden layer neural networks. Two of these advancements, dropout and batch normalization, have shown significant promise in deep learning models. Let's begin with a discussion of dropout and then examine batch normalization.

Dropout adds noise to the learning process so that the model is more generalizable. Training an ensemble of deep neural networks with several hundred thousand parameters each might be infeasible. As seen in Figure 1.9, dropout adds noise to the learning process so that the model is more generalizable.

Figure 1.9: Regularization Techniques



The goal of dropout is to approximate an ensemble of many possible model structures through a process that perturbs the learning in an attempt to prevent weights from co-adapting. For example, imagine we are training a neural network to identify human faces, and one of the hidden units used in the model sufficiently captures the mouth. All other hidden units are now relying, at least in some part, on this hidden unit to help identify a face through the presence of the mouth. Removing the hidden unit that captures the mouth forces the remaining hidden units to adjust and compensate. This process pushes each hidden unit to be more of a "generalist" than a "specialist" because each hidden unit must reduce its reliance on other hidden units in the model.

During the process of dropout, hidden units or inputs (or both) are randomly removed from training for a period of weight updates. Removing the hidden unit from the model is as simple as multiplying the unit's output by zero. The removed unit's weights are not lost but rather frozen. Each time that units are removed, the resulting network is referred to as a *thinned network*. After several weight updates, all hidden and input units are returned to the network. Afterward, a new subset of hidden or input units (or both) are randomly selected and removed for several weight updates. The process is repeated until the maximum training iterations are reached or the optimization procedure converges.

In SAS Viya, you can specify the DROPOUT= option in an ADDLAYER statement to implement dropout. DROPOUT=*ratio* specifies the dropout ratio of the layer.

Below is an example of dropout implementation in an ADDLAYER statement.

```
AddLayer/model='DLNN' name="HLayer1" layer={type='FULLCONNECT' n=30
         act='ELU' init='xavier' dropout=.05} srcLayers={"data"};
```

Note: The ADDLAYER syntax is described shortly and further expanded upon throughout this book.

Batch Normalization

The *batch normalization* (Ioffe and Szegedy, 2015) operation normalizes information passed between hidden layers per mini-batch by performing a standardizing calculation to each piece of input data. The standardizing calculation subtracts the mean of the data and then divides by the standard deviation. It then follows this calculation by multiplying the data by the value of a learned constant and then adding the value of another learned constant.

Thus, the normalization formula is

$$\gamma * (\frac{X_i - \mu}{\sigma}) + \beta$$

where gamma (γ) and beta (β) are learnable parameters.

Some deep learning practitioners have dismissed the use of sigmoidal activations in the hidden units. Their dismissal might have been premature, however, with the discovery of batch normalization. Without batch normalization, each hidden layer is, in essence, learning from information that is constantly changing when multiple hidden layers are present in a neural network. That is, a weight update is reliant on second-order, third-order (and so on) effects (weights in the other layers). This phenomenon is known as the *internal covariance shift* (ICS) (Ioffe and Szegedy, 2015).

There are two schools of thought as to why batch normalization improves the learning process. The first comes from Ioffe and Szegedy who believe batch normalization reduces ICS. The second comes from Santurkar, Tsipras, Ilyas, and Madry who argue that batch normalization is not really reducing ICS but is instead smoothing the error landscape (Santurkar, Tsipras, Ilyas, and Madry 2018). Regardless of which thought prevails, batch normalization has empirically shown to improve the learning process and reduce neuron saturation.

In the SAS deep learning actions, batch normalization is implemented as a separate layer type and can be placed anywhere after the input layer and before the output layer.

Note: With regard to convolutional neural networks, the batch normalization layer is typically inserted after a convolution or pooling layer.