SAS® Visual Data Mining and Machine Learning 8.2: Deep Learning Programming Guide

SAS® Documentation
December 14, 2017

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS® Visual Data Mining and Machine Learning 8.2: Deep Learning Programming Guide*. Cary, NC: SAS Institute Inc.

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

December 2017

8.2-P2:casdlpg

Contents

1

What's New in SAS Deep Learning

What's New in SAS Deep Learning

SAS Deep Learning Toolkit on SAS Viya 3.3

The SAS Deep Learning toolkit is a new set of cloud-enabled SAS CAS actions released with SAS Viya 3.3. The actions are delivered as part of SAS Visual Data Mining and Machine Learning (VDMML) 8.2. The new Deep Learning toolkit includes the following key features:

- Supports deep fully-connected neural networks (DNN), convolutional neural networks (CNN), and recurrent neural networks (RNN).

  Note: In this documentation, DNN always refers to a deep fully-connected neural network.
- Provides layer-based operations to build customized neural networks.
- Accepts raw images, text, and numeric data as input types.
- Supports output of each layer of the network.
- Supports regression, classification, autoencoder, and seq2seq modeling tasks.
- Provides Dropout, L1, and L2 regularizations to prevent model overfitting.
- Utilizes modern Stochastic Gradient Descent (SGD) and Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) optimization methods.
- Supports AStore generations for deep neural networks (DNN) and convolutional neural networks (CNN), as well as DS1 score codes for DNN.
- Supports importing trained DNN and CNN Caffe models.
- SAS Deep Learning actions support Massively Parallel Processing (MPP) and Symmetric Parallel Processing (SMP) with multiple threading.
- Supports for one or more GPU processors with SMP servers.
- Supports customized activation and loss functions using SAS FCMP (Function Compiler).
- Supports automatically tuning optimization hyperparameters.

New Deep Learning Action Sets

The following SAS Deep Learning Action sets are new:

- The deepNeural action set enables you to model and score tasks that require deep fully-connected neural networks.
- The deepRnn action set enables you to model and score tasks that require deep recurrent neural networks, using text as the input.
- The deepLearn action set enables you to model and score tasks that require deep learning networks. The deepLearn action supports DNN, CNN, and RNN.

GPU Support Information

The SAS Deep Learning toolkit provides GPU support under the following conditions:

- GPU support is only enabled for SMP architectures.
- GPU is supported for feed forward neural network architectures with fully connected and convolutional layers, such as DNN and CNN.
- Only one GPU is supported when using a batch normalization (BN) layer.
- The layerOut specification to generate the feature maps for selected layers is not supported with dlScore.
- GPU does not support Astore scoring, FCMP, or L-BFGS.
- The **nThreads=** value must be greater than or equal to the number of available GPU devices in order to use all of them.
- GPU support for the batch normalization (BN) layer is experimental.
- Sharing GPU among multiple processes is not recommended. GPUs need to be used exclusively.

2

Deep Learning Technical Concepts

Convolutional Neural Networks

Overview

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 is composed of multiple layers and a number of neurons. CNNs are designed to take image data as input. This assumption allows CNNs to use structural information so that they are more computational efficient and use less memory.

Layers in CNNs

A typical CNN consists of four types of layers: input layer, convolution layer, pooling layer, and output layer. Each type of layer has its own specific properties and functionalities. Below is a brief description of those layers.

Input Layer
: The input layer stores the raw pixel values of the image to be classified. For instance, each image in the CIFAR-10 data set has a height of 32, width of 32, and three color channels of blue, green, and red.

Convolution Layer
: The convolution layer computes the output of neurons that are connected to local regions in the input. Each computes a dot product between their weights and a small region where they are connected to the input volume. If you decide to use 12 filters, a volume might result such as 32 x 32 x 12.

  The convolution layer parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a CNN might have size 5 x 5 x 3, that is, 5 pixels width, 5 pixels height, and 3 filters (blue, green, and red). During the forward pass, the algorithm convolves (slides) each filter across the width and height of the input volume and computes dot products between the entries of the filter and the input at any position. As the filter moves over the width and height of the input volume we produce a 2-dimensional activation map that gives the responses of that filter at every spatial position.

  Intuitively, the network learns filters that activate when they see some type of visual feature (such as an edge of some orientation, or a blotch of some color) on the first layer. Eventually some patterns are detected on higher layers of the network.

When building a CNN using SAS Deep Learning actions, the researcher does not need to specify padding for the convolution (or pooling) layer. Padding is automatically calculated so that, for a stride of 1, the image size after convolution is exactly the same as it was before convolution. For a stride of 2, the output image size is 1/2 the input image size, and so on.

For more information about how SAS Deep Learning performs convolution layer padding, see Automatic Padding Calculations for Convolution Layers on page 6.

Pooling Layer

The pooling layer performs a downsampling operation along the spatial (height, width) dimensions.

It is common to periodically insert a pooling layer in-between successive convolution layers in a CNN architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The pooling layer operates independently on every depth slice of the input and resizes it spatially, for instance, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged.

In addition to max pooling, the pooling units can also perform other functions, such as global average pooling or mean pooling.

Output Layer

The output layer is essentially a fully connected layer that is associated with a particular loss function. The output layer computes the class scores, with a resultant volume of size 1 x 1 x 10, such that each of the 10 digits in the third dimension corresponds to a class score, where class scores map to the 10 categories of the CIFAR-10 image set.

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular neural networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

These processes are how CNNs transform original images layer by layer, from the original pixel values, to the final class scores. You might observe that some layers contain parameters and some layers do not. For instance, the convolution and fully connected layers perform transformations that are not only a function of the activations in the input volume, but also of the parameters (the weights and biases of the neuron connections). In contrast, the pooling layer implements fixed functions. The parameters in the convolution and fully connected layers are trained using stochastic gradient descent or the L-BFGS optimizer. This means that the class scores the CNN computes will be consistent with the labels in the training set for each image.

Neurons in CNNs

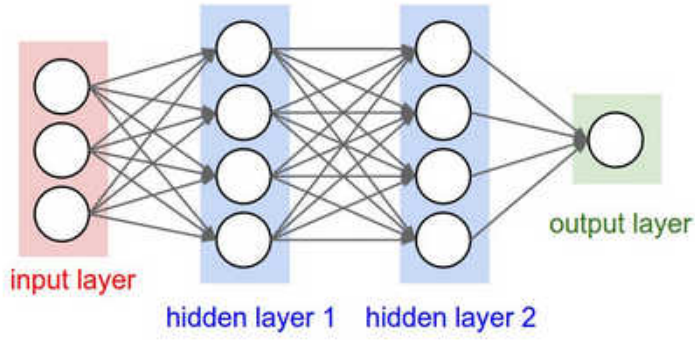
CNNs take advantage of the fact that the input consists of images. As a result, CNN network layers constrain the architecture more sensibly. Unlike an ordinary neural network, the layers in a CNN have 3 dimensions: width, height, and depth.

Note: The word depth here refers to the third dimension of an activation volume, not to the depth of a full neural network.

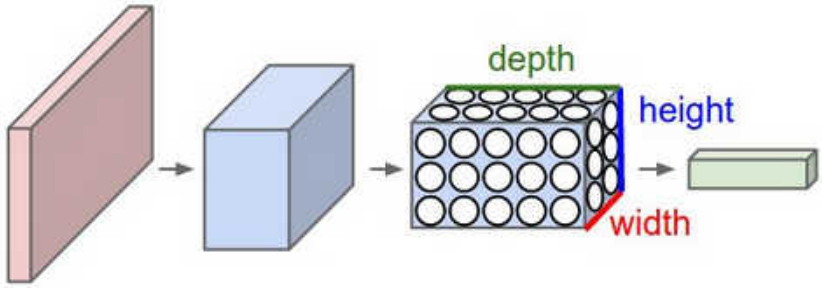
Consider the input images in Krizhevsky, Nair, and Hinton's CIFAR-10 data set. The data set contains 60,000 32x32 color images distributed across 10 classes, with 6000 images per class.

In other words, the CIFAR-10 images are an input volume of activations, and the volume has dimensions of 32 x 32 x 3 (width in pixels, height in pixels, depth in channels, respectively). The neurons in a layer will be connected only to a small region of the layer before it, instead of all of the neurons in a fully connected manner. It follows that the final output layer for the CIFAR-10 data using CNN would have dimensions of 1 x 1 x 10, because by the end of the CNN architecture, the full image is reduced into a single vector of class scores, arranged along the depth dimension.

Here is a visual representation of a regular 3-layer neural network:



Here is a visual representation of a CNN model.



Note: Image credit: "CS231n: Convolutional Neural Networks for Visual Recognition (Spring 2017)," A. Karpathy, Stanford University. http://cs231n.github.io/convolutional-networks/#add

You can see that the CNN arranges its neurons in three dimensions (height, width, depth) as visualized in one of the layers. In a CNN, every layer transforms the 3-dimensional input volume to a 3-dimensional output volume of neuron activations with three convolution filters. Here, the pink input layer holds the image. The image height (32) and width (32) are specified in the corresponding dimensions using pixels. The depth is 3, for the three (blue, green, and red) filters.

Automatic Padding Calculations for Convolution Layers

Currently, SAS Deep Learning action users cannot specify the padding to be used by a convolution layer. In order to be able to match the padding used by other frameworks, that feature may be added in the future.

SAS Deep Learning calculates automatic padding for a convolutional layer as follows:

The padding is calculated with the goal of producing an output image in a size that will be "reasonable", based on the original input image size and filter size. The key issues are whether the input image dimensions are even or odd, whether the filter dimensions are even or odd, and the value of the user-specified stride.

Some of the "reasonable" rules are:

- The horizontal and vertical padding sizes are independent of each other. Changing the horizontal dimension of an image or filter will have no effect on the vertical padding size.
- If the stride is 1, then the output image size is the same as the input image size. When discussing convolutions, this is sometimes referred to as "same padding".
- If the stride is 2, then the output image size is about one half the input image size.
- Padding is first added to the right side of the image, then the left. So, if padding is unequal, the right side will have more padding than the left.
- Images are padded with zeros.

In order to match the padding calculations between CPU and GPU, you should specify the **forceEqualPadding** option when you call the dlTrain action.

Common CNN Architectures

Here are some of the most commonly referred to convolutional neural networks that have named architectures:

Lenet
: The first successful applications of convolutional networks were developed by Yann LeCun in 1990s. Of these, the best known is the LeNetarchitecture. The LeNet architecture was used to read ZIP codes, digits, etc.

AlexNet
: The AlexNet was the first work that popularized convolutional networks in computer vision, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The AlexNet was submitted to the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 and significantly outperformed the second runner-up: for instance, AlexNet had a Top 5 error of 16%, whereas the runner-up had a Top 5 error of 26%. AlexNet has a very similar architecture to LeNet, but is deeper, bigger, and features convolutional layers stacked on top of each other. This is significant because beforehand, it was common for architectures to have only a single convolutional layer, which was always immediately followed by a pooling layer.

ZF Net
: Matthew Zeiler and Rob Fergus won the 2013 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) using a convolutional network architecture called the ZFNet. It improved performance over AlexNet by tuning the architecture hyperparameters, by expanding the size of the middle convolutional layers and, by using a smaller stride and filter size on the first layer.

GoogLeNet
: The 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winner was a convolutional network from Szegedy et al. from Google. Its main contribution was the development of an inception module that dramatically reduced the number of parameters in the network (4M, compared to 60M in AlexNet). This architecture uses average pooling instead of fully connected layers at the top of the convolutional network. This improvement eliminates a large amount of parameters that do not seem to matter much.

VGGNet
: The 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) runner-up was the network from Karen Simonyan and Andrew Zisserman. Their network was called VGGNet. VGGNet showed that the depth of a convolutional network is a critical component for good performance. Their final best network contains 16 convolutional/fully-connected layers with a homogeneous architecture that performs only 3x3 convolutions and 2x2 pooling from the beginning to the end. A pretrained VGGNet model is available for plug and play use in Caffe. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it was since found that these fully connected layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.

ResNet
: ResNet is a residual network developed by Kaiming He et al. ResNet was the winner of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2015. ResNet features special skip connections and a strong use of batch normalization. The architecture also notable because it does not have any fully connected layers at the end of the network. ResNets are considered by many (in 2016) to be state of the art convolutional neural network models.

The SAS Deep Learning toolset delivered with SAS Viya 3.3 supports VGG-like and ResNet-like CNN models .

Batch Normalization

Overview

Batch Normalization is an optimization method that often results in faster convergence (in terms of number of epochs required) and better generalization of deep neural networks. The February 2015 paper, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," by Sergey Ioffe and Christia Szegedy describes the rationale and implementation of batch normalization in deep neural networks.

In the SAS Deep Learning actions, batch normalization is implemented as a separate layer type. The batch normalization layer is typically inserted after a convolution or pooling layer, but batch normalization layers can be placed anywhere after the input layer and prior to the output layer.

Batch Normalization Formula

The batch normalization operation normalizes a set of input data 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:

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

Gamma and beta are learnable parameters. When training, the mean and standard deviation are computed over the current mini-batch of the training data set. When scoring, the mean and standard deviation used are those computed over the entire training data set. In addition, if the data from source layer to the batch normalization layer is structured as feature maps (as in the convolution layer), then a separate mean, standard deviation, gamma, and beta is computed for each feature map (i.e. the mean of a feature map is the average value of ALL pixels in the feature map for each feature map in the mini-batch). If the data from the source layer is not structured as feature maps (as in the fully connected layer), then a separate mean, standard deviation, gamma, and beta is computed for each neuron.

Source Layer Parameter Settings

In order for the batch normalization computations to conform to those described in Sergey Ioffe and Christian Szegedy's Batch Normalization research, the source layer should have settings of `act=identity` and `includeBias=False`. The activation function that would normally have been specified in the source layer should instead be specified on the batch normalization layer. If you do not configure your model to follow these option settings, the computation will still work, but it will not match the computation as described by Ioffe and Szegedy.

Computing Related Statistics when Scoring

The statistics used in the batch normalization formula (that is, the mean and standard deviation for each feature map or neuron) are computed during training, and then re-computed over each mini-batch. When scoring, however, it is more appropriate to use mean and standard deviation statistics that are computed over the entire training data set. In order to perform this task without making an extra pass through the data at the end of training, the SAS Deep Learning actions accumulate the sum(x) and sum(x^2) for each feature map or neuron over an entire epoch. Then those sums are used to compute the mean and variance over the entire training data set, using the standard mean and variance computing formulas.

Of course, these are not quite the true mean and standard deviation over the entire training data set. This is because the weights are adjusted after each mini-batch, which means that the sums are computed using different weights. However, in the last epoch of most training runs, those weights often do not change much

when the neural network is converged well, or if the learning rate is small in SGD.Experiments have shown that if the training has converged significantly (to the point where the weights are no longer changing much), the resulting final means and standard deviation sums are reasonably close to the actual means and standard deviation calculated over the entire training data set using a fixed set of weights. Experiments have also shown that the accuracy of scoring is not affected by small variations in mean and standard deviation measurements.

Storing Trained Batch Normalization Parameters

When using SAS Deep Learning actions, the batch normalization parameters are saved as part of the weight table. The learnable batch normalization layer parameters, gamma and beta, for each feature map or neuron, are saved in the bias weight portion of each batch normalization layer's weights. That is so that they will not be part of the L1 or L2 regularization computations. The batch normalization statistics (mean and standard deviation for each feature map or neuron) for each batch normalization layer are stored at the end of the weight table.

Current Limitations

Currently, SAS Deep Learning batch normalization is experimental on GPU, and is supported only on a single GPU with the parameter specification `nThreads=1`. Batch normalization is supported on multiple CPUs with any value of `nThreads=`. Batch normalization on CPUs supports only synchronous SGD and L-BFGS. The statistics required by batch normalization are computed for each mini-batch, which slows down the training process. Future releases of the SAS Deep Learning tools will improve the scalability and speed performance of batch normalization.

References

- “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, Sergey Ioffe and Christian Szegedy (2015), International Conference on Machine Learning, pp. 448–456, https://arxiv.org/abs/1502.03167
- “CS231n: Convolutional Neural Networks for Visual Recognition” (2017), Andrej Karpathy, Stanford University, Stanford CA. http://cs231n.github.io/convolutional-networks/.
- Images ”Fully Connected Neural Network” and “Convolutional Neural Network” from CS231n: Convolutional Neural Networks for Visual Recognition” (2017), Andrej Karpathy, Stanford University, Stanford CA. http://cs231n.github.io/convolutional-networks/.
- “Gradient-Based Learning Applied to Document Recognition”, Yann LeCun, Leon Bouttou, Yoshua Benigo, and Patrick Haffner, Procedures of the IEEE, November 1988, http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf.
- "Deep Residual Learning for Image Recognition", Kaiming He et. al, (2015), https://arxiv.org/abs/1512.03385.
- "Very Deep Convolutional Networks for Large Scale Visual Recognition”, Visual Geometry Group, University of Oxford, Oxford England. (2014) http://www.robots.ox.ac.uk/~vgg/research/very_deep/.
- “Going Deeper with Convolutions”, Szegedy et al., (2014) Google, https://arxiv.org/abs/1409.4842.
- “Visualizing and Understanding Convolutional Networks”, M. Zeiler and R. Fergus, (2013) https://arxiv.org/abs/1311.2901.
- "ImageNet Classification with Deep Convolutional Neural Networks", Neural Information Processing Systems (NIPS), A. Krizhevsky, I. Sutskever, and G. E. Hinton, http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

Recurrent Neural Networks

Overview

Recurrent Neural Networks (RNNs) are specifically designed to handle sequence data, such as speech, text, time series, and so on. RNNs are called recurrent because they perform the same task for every element of a sequence. The output for each element depends on the computations of its preceding elements.

The original RNN is quite simple in architecture, but can be very hard to train when sequences get long. Two popular variants of RNN are popular: Long Short-Term Memory (LSTM) and Gate Recurrent Unit (GRU). The SAS Deep Learning actions support all three model types (RNN, LSTM, GRU).

The formulas used for the Deep Learning RNN, LSTM, and GRU algorithms can be seen and found in Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, by Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio, Université de Montreal, 2014.

Peephole Connections in RNNs

Peephole connections were created to improve performance of Long Short-Term Memory (LSTM) networks. Peepholes connect the LSTM memory cell to non-linear gates (input, output, forget) that regulate the flow of signals in and out of the cell. This behavior allows the gates in LSTM networks to not only depend on the hidden state, s_{t-1}, but also on the previous internal state c_{t-1}. This adds an additional term to the gate equations. Peephole connections can effectively regulate long range dependencies and improve RNN training performance.

The current SAS Deep Learning implementation of LSTM uses peephole connections that are always turned on. Future updates of the SAS Deep Learning actions will feature configurable peephole connection settings.

Typical RNN Applications

RNNs that you build using SAS Deep Learning actions support the following application types:

- **Forecasting and time series:** applications where the input is numeric sequence data, and the output is a single numeric value or a nominal target value.
- **Sentiment analysis and text catergorization:** applications where the input is text data, and the output is a single numeric value or a nominal target value.
- **Automatic speech recognition:** applications where the input is a numeric sequence, and the output is nominal labels for the input sequence. Only CTC loss is supported in this application.
- **Text summarization, and simple question and answer (Q&A):** applications where the input is text, and the output is also text.

Supported Major Features

The SAS Deep Learning tools set supports the following algorithms and statistical tools for RNNs:

- Sampled Softmax for text generation with very large vocabulary.
- Bidirectional recurrent layers.
- Connectionist Temporal Classification (CTC) loss function for speech recognition.
- Text tokenization and term-to-embedding mapping: text can be used as direct input and output, significantly simplifying text tasks.

Using Text Generation Models

The following usage notes can be helpful when using deep learning text generation models:

- When you perform text generation, make sure that you store the word embeddings with ID mapping during training, and then use the same table for scoring. If you do not use the same table for scoring, your results will be invalid.
- Word embeddings are numeric vector representations of words trained based on large corpus, typically without any label information. Popular word embeddings include Word2Vec, Glove and FastText, and they are publicly available for download.

  Any column name can be used in an embedding table, but the first column of the embedding table *must* be the word, and the rest of the columns are treated as numeric vector representation of that word.

  SAS Deep Learning tools currently do not support on-the-fly task-oriented word embedding training. SAS deep learning experiments and other public research papers show that task-oriented word embedding training does not always lead to better results, unless a huge amount of labeled data is available. Using pre-trained word embeddings like Word2Vec, Glove or FastText should be sufficient in most cases. SAS Deep Learning tools will provide embedding training capability in the future.
- Use the modelOutputEmbeddings table for output embeddings, and use the *original* embedding table for input embeddings. This is because the modelOutputEmbeddings table contains only terms that are present in output text, and discards terms that are present only in the input text. If you use the modelOutputEmbeddings table for input embeddings, your results tend to contain more unknown terms.
- Make sure you specify values for `hasInputTermIds` and `hasOutputTermIds` (if applicable) correctly for input and output embedding tables.
- Training and scoring for text generation are slightly different. Training uses the *ground truth* word to predict the next word, while scoring uses the *predicted word* to predict the next word. Because of the difference, you should expect the reported error rates for training and scoring to be different.
- When you perform text generation with sampled Softmax, training and scoring are different. Training uses a small negative sample set to speed up training, while scoring considers all words in the dictionary. Because of this difference, you should expect the results for training and scoring to be different.
- Use sampled Softmax only when the number of levels is extremely large, for example, greater than 10,000.
- Arbitrary length layers should be used in a text generation model.

Using Sequence Analysis Models

The following usage notes are useful when using deep learning sequence analysis models:

- When you perform numeric sequence analysis, make sure that you specify the correct token size (how many numbers per step). The default token size value is 1, which might be incorrect in some data sets.
- When you perform numeric sequence analysis for sequences that have different lengths, make sure that there is a column that stores the actual sequence length for the row, and specify the sequence length parameter with the column name at training time. Otherwise, your results will be incorrect.
- When you specify a target sequence, you must use the CTC loss function. Other loss functions are not supported for target sequences.

Using Generic RNN Models

The following usage notes are useful when using generic deep learning RNN models:

- If your model generates floating point exception errors, consider specifying values for the gradient clipping parameters (`clipGradMin` and `clipGradMax`). Once unrolled, RNN models can be extremely deep. Very deep RNN models can generate floating point overflow conditions.
- GRU models usually gets similar results as LSTM models, but GRU models train faster, and GRU models are smaller in size.
- In order to build a decent text generation model, a lot of data is required and a very long training is needed.
- For best deep learning results, always shuffle your input data before training, and whenever possible, use sampling techniques to equalize class populations across the input training data.
- Generally, different use cases require different model architectures. For instance, while text generation needs an arbitrary length layer, classification requires an encoding layer.

References

- "Learning precise timing with LSTM recurrent networks," F. A. Gers, N. N. Schraudolph, and J. Schmidhuber (2003). Journal of Machine Learning Research, vol. 3, pp. 115-143, 2003.
- "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling", J. Chung, C. Gulcehre, K. Cho, and Y. Bengio (2014), Université de Montreal, http://arxiv.org/pdf/1412.3555v1.pdf.

Using the SAS Function Compiler (FCMP)

Overview

The SAS Deep Learning action set provides basic activation and error functions, but you can create your own custom activation and error functions using the SAS Function Compiler (FCMP). The FCMP procedure is part of Base SAS software. You use FCMP to create user-defined functions that can be used in a SAS DATA step (as well as in many SAS procedures). You can use FCMP with SAS Deep Learning to provide custom error and activation functions for deployment to individual vector components.

After you define your activation and error functions using FCMP, you must specify the libraries that you need. You do this by using the OPTIONS statement to specify CMPLIB= system options. For more information about the OPTIONS statement, see "SAS Global Statements: Reference" in SAS Global Statements: Reference .SAS Global Statements: Reference. For more information about the CMPLIB= system option, see SAS System Options: Reference.

Note: Because FCMP is applied to individual vector components (instead of an entire vector), your gradient calculations will be incorrect if you use FCMP in conjunction with the Softmax activation function. In general, Softmax should be used only with the supported (non-FCMP equivalent) cross-entropy error function.

Here is a simple example that shows how FCMP can be used in conjunction with the SAS Deep Learning actions. The purpose of this example is to demonstrate suggested syntax for defining activation and error functions using FCMP.

Note: The trained model in the example below is not fully converged. The model can be further tuned and optimized, but the intent of this example is to demonstrate how to define and use SAS FCMP activation functions and error functions. Tuning and optimizing the example model below might be an interesting individual exercise for experienced users.

Load the Fcmpact Action Set and Specify the Session Library

In this Python example, we connect to a CAS server by importing the SAS Scripting Wrapper for Analytics Transfer (SWAT) and using the `sw.cas` package to create a connection. Then we load the fcmpact action set and set the session CMPLIB= system option to specify the function library to reference:

```
import swat as sw

s = sw.CAS('server-name.unx.my-org.com', 35131, nworkers=1)
s.sessionprop.setsessopt(caslib='CASUSER',timeout=31535000)
s.loadactionset('image')
s.loadactionset('deeplearn')
s.loadactionset('fcmpact')
s.sessionProp.setsessopt(cmplib='CASUSER.myactdef')
```

Here is the response:

```
NOTE: 'CASUSER(userID)' is now the active caslib.
NOTE: Added action set 'image'.
NOTE: Added action set 'deeplearn'.
NOTE: Added action set 'fcmpact'.
```

Define Functions Using the AddRoutines Command

Next we use the AddRoutines command to define a self-gated activation function called `swish`, and an error function called `error`. The functions create a smoothed approximation to the sum of absolute values between the model output vector and the target vector. The activation function routines are saved to a table named `myactdef`. The cmplib library name is `myactdef`.

Note: Functions that are defined by FCMP are applied to individual elements in the applicable layer vector, instead of the entire vector. You should not expect to be able to write a function that takes matrix inputs and operates on matrices.

```
s.addRoutines(
routineCode = {'''
    function swish( t );
        if (t >= 0.0) then ft =  1.0 / (1.0 + exp(-t));
        else do;
            z = exp(t);
            ft = z / (1.0 + z);
        end;
        return(.3*t*ft);
    endsub;
    ''','''
    function softplus( t );
        if (t <= 0.0) then ft =  log(1.0 + exp(t));
        else do;
            ft = t + log(1+exp(-t));
        end;
        return(ft);
    endsub;
   ''','''
    function error(t, target);
        err = (t-target);
```

```
        err = softplus(err) + softplus(-err) - 2*softplus(0);
        return(err);
    endsub;
    ''' }   ,
    package = "pkg",
    saveTable=1,
    funcTable = dict(name="myactdef",caslib="casuser", replace=1)
);
```

Load CIFAR-10 Train and Test Data Sets

CIFAR-10 is an established computer-vision data set used for object recognition. It is a labeled subset of the 80 Million Tiny Images data set, and consists of 60,000 32x32 color images. Each color image contains one of 10 object classes, with 6000 images per class.

Here we load the CAS training data, `cifar10_train_sample.sas7bdat`, and the CAS test data, `cifar10_test.sas7bdat`, into the CAS session.

```
s.table.loadtable(
   path='cifar10_train_sample.sas7bdat')

s.table.loadtable(
   path='cifar10_test.sas7bdat')
   testTbl = s.CASTable(
        'cifar10_test',
         caslib='CASUSER')
   shuffleidSASCode = "call streaminit(__rankid*1000);
   shuffle_id=rand(\"UNIFORM\");"

trainTbl = s.CASTable(
   name='cifar10_train_sample',
   computedVars=['shuffle_id'],
   computedVarsProgram=shuffleidSASCode,
   groupBy='shuffle_id');

info=s.partition(
   table=trainTbl,
   casout=dict(
       name='cifar10_train_shuffle',
       replace=True,
       blocksize='32')
);
```

The response is:

```
NOTE: Cloud Analytic Services made the file cifar10_train_sample.sas7bdat
       available as table CIFAR10_TRAIN_SAMPLE in caslib CASUSER(userID).
NOTE: Cloud Analytic Services made the file cifar10_test.sas7bdat
      available as table CIFAR10_TEST in caslib CASUSER(userID).
```

Create Models and Input Layer

This step creates a convolutional network model. We create the input layer, which holds the raw pixel values of the image. The input layers are 32 pixels wide, 32 pixels high, with 3 channels (RGB), or 32 x 32 x 3. RGB values are offset by the provided channel means of 126.02464141 (R), 123.7085042 (G), and 114.85431865 (B).

```
channel_means = [126.02464141, 123.7085042, 114.85431865]

s.buildmodel(
   model=dict(
         name='FCMPNet',
         replace=True),
   type='CNN')

s.addLayer(
   model='FCMPNet',
   name='data',
   layer=dict(
         type='input',
         nchannels=3,
         width=32,
         height=32,
         scale=1,
         offsets = channel_means)
 ) ;
```

Create a Swish Convolution Layer

This step adds convolution layers to the FCMPNet model. The convolutional layer computes the output of neurons that are connected to the local regions in the input. The output is the dot product of individual neuron weights and the small region that they are connected to in the input volume. The convolution layers use the swish activation function that was defined earlier in the example. The kernel is 3 x 3, the step is 1, and `nFilters`=32. Initial weights are specified in `xavier`.

```
s.addLayer(
   model='FCMPNet',
   name='conv1',
   layer=dict(
         type='convolution',
         fcmpact="swish",
         nFilters=32,
         width=3,
         height=3,
         stride=1,
         init="xavier"),
         srcLayers=['data']
     );

s.addLayer(
    model='FCMPNet',
    name='conv2',
    layer=dict(
          type='convolution',
          fcmpact="swish",
          nFilters=32,
          width=3,
          height=3,
          stride=1,
          init="xavier"),
          srcLayers=['conv1']
```

```
    );
```

Add a Pooling Layer

This step adds a pooling (downsampling) layer. The downsampling takes place along the height and width dimensions to result in a lower volume. Here we use a 5 x 5 kernel, a stride of 1, and we perform the pooling operation using mean values. Pooling layers do not support activation function definitions.

```
s.addLayer(
   model='FCMPNet',
   name='pool1',
   layer=dict(
         type='pooling',
         width=5,
         height=5,
         stride=1,
         pool='mean'),
         srcLayers=['conv2']
   );
```

Add Additional Convolution and Pooling Layers

This step adds two more convolutions layers and one more pooling layer. The convolutional layers in his step use 5 x 5 kernels, as opposed to the 3 x 3 kernel that was used in the initial convolutional layer.

```
s.addLayer(
   model='FCMPNet',
   name='conv3',
   layer=dict(
         type='convolution',
         fcmpact="swish",
         nFilters=32,
         width=5,
         height=5,
         stride=1,
         init="xavier"),
         srcLayers=['pool1']
   );

s.addLayer(
   model='FCMPNet',
   name='conv4',
   layer=dict(
         type='convolution',
         fcmpact="swish",
         nFilters=32,
         width=5,
         height=5,
         stride=1,
         init="xavier"),
   srcLayers=['conv3']
   );

s.addLayer(
```

```
   model='FCMPNet',
   name='pool2',
   layer=dict(
         type='pooling',
         width=5,
         height=5,
         stride=1,
         pool='mean'),
   srcLayers=['conv4']
   );
```

Add an Output Layer to Compute Class Scores

The final fully connected layer in a convolutional neural network (CNN) is the output layer. The output layer computes the class scores. For the CIFAR-10 data, the output layer has dimensions of 1 x 1 x 10, reducing the full image into a single vector of class scores, arranged along the depth dimension. Each of the 10 numbers corresponds to a class score, in this case, the 10 categories associated with the CIFAR-10 images.

In this example, we use the FCMP error function called `error` that we defined earlier using the AddRoutines command on page 13. The first argument of the error function is treated as the predicted value, and the second argument is treated as the corresponding target.

```
s.addLayer(
   model='FCMPNet',
   name='outlayer',
   layer=dict(
         type='output',
         fcmperr='error',
         fcmpact='swish',
         init='xavier'),
   srcLayers=['pool2']
   );
```

Train the CNN and FCMP Deep Neural Models

After you specify customized error and activation functions, identify your training data, and configure your deep neural model, you are ready to train the model. In this instance, the image is expanded into a series of 32*32*3 columns. Each column corresponds to a pixel. The index starts from the top left corner and increases row by row and channel by channel (BGR).

```
inputVars = []
for i in range(1,3073):
    inputVars.append('var'+str(i));

r = s.dltrain(
   model='FCMPNet',
   table=trainTbl,
   seed=54323,
   target='labels',
   nominal=['labels'],
   inputs=inputVars,
   validTable=testTbl,
   modelWeights=s.CASTable(
        'FCMPNet_Weights',
        replace=True),
```

```
   optimizer=dict(
        algorithm=dict(
           method='adam',
           lrPolicy='step',
           stepsize=2,
           learningrate=.0005,
           gamma=.8),
    maxEpochs=10,
    minibatchsize=1,
    loglevel=2,
    regL2=1e-4)
   );
```

Here is the response:

```
NOTE:  The Synchronous mode is enabled.
       NOTE:  The total number of parameters is 92074.
       NOTE:  The approximate memory cost is 85.00 MB.
       NOTE:  Initializing each layer cost 0.44 (s).
       NOTE:  The total number of workers is 1.
       NOTE:  The total number of threads on each worker is 32.
       NOTE:  The total number of minibatch size per thread on each worker is 1.
       NOTE:  The maximum number of minibatch size across all workers for the
              synchronous mode is 32.
       NOTE:  Target variable: labels
       NOTE:  Number of levels for the target variable: 10
       NOTE:  Levels for the target variable:
       NOTE:  Level      0:            5
       NOTE:  Level      1:            4
       NOTE:  Level      2:            7
       NOTE:  Level      3:            6
       NOTE:  Level      4:            1
       NOTE:  Level      5:            0
       NOTE:  Level      6:            3
       NOTE:  Level      7:            2
       NOTE:  Level      8:            9
       NOTE:  Level      9:            8
       NOTE:  Number of input variables:  3072
       NOTE:  Number of numeric input variables:   3072
       NOTE:  Epoch       Learning Rate     Loss    Fit Error    Time (s)
       NOTE:          0       0.001        17772     0.8949       9.97
       NOTE:          1       0.001       33.391     0.8951       9.87
       NOTE:          2       0.001       29.572     0.8951      10.02
       NOTE:  The optimization reached the maximum number of epochs.
       NOTE:  The total time is 29.86 (s).
```

Score FCMPNet Model

The output scores for the FCMPNet models summarize the number of observations read and used, the misclassification error, and the loss error for of each model. SAS Deep Learning uses `dlscore` to perform this task.

Here is the code to score the model FCMPNet:

```
s.dlscore(
   model='FCMPNet',
```

```
    initWeights='FCMPNet_Weights',
    table=testTbl );
```

Here is the response:

Table 2.1 *FCMPNet Model Scores*

	Description	Value
0	Number of Observations Read	10000
1	Number of Observations Used	10000
2	Misclassification Error (%)	45.17
3	Loss Error	0.159988

Deploying the Model with FCMP Codes

The deep learning model stores the names of the FCMP functions, but not the function definitions themselves. If you deploy the model that you create in a new session, you must submit your FCMP function definitions to the new session. Another way is to save and upload your FCMP library table to the new session. Remember to specify any custom CMPLIB= session options that you declared with the original model.

Note: If the only dependency on FCMP comes from the error function definition, then you can score observations *as long as the target variable is not present*. In this special case, FCMP would not be needed. Currently, deep learning models with FCMP definition dependencies are not supported for DS1 and ASTORE.

Jupyter Notebook for the FCMPNet Example

Here is the Jupyter Notebook for this example.

References

CIFAR-10 Dataset: A. Krizhevsky, V. Nair, and G. Hinton, University of Toronto (2012), https://www.cs.toronto.edu/~kriz/cifar.html.

80 Million Tiny Images Dataset: A. Torralba, R. Fergus, W. Freeman, Massachusetts Institute of Technology (2008), http://groups.csail.mit.edu/vision/TinyImages/.

Stochastic Gradient Descent Optimizer

Overview

Gradient descent is one of the most popular optimization algorithms for neural networks. Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ (with respect to the parameters). The learning rate η determines the size of the steps taken to reach a (local) minimum. In other words, gradient descent optimizations follow the direction of the slope of the surface (created by the objective function) downhill until a valley or saddle point is reached.

Unlike gradient descent, which computes the gradient of the objective function with respect to the model parameters for the entire training set, Stochastic Gradient Descent (SGD) computes gradients for small subsets of the training data, called mini-batches. The SGD algorithm then updates the model parameters for each mini-batch. Computing the gradient for a mini-batch requires much less computational time than computing the gradient on the entire training set. This enables SGD to perform many updates on the model parameters in the amount of time required for gradient descent to perform a single update. However, since each parameter update is based on only a mini-batch, there is no guarantee that each update improves the objective value over the entire training set.

In practice, SGD can perform well for machine learning problems. There are several methods that exist to improve the performance of SGD that are available in the Deep Learning action set.

- The momentum method accumulates parameter updates in a velocity vector, which is dampened after each update.
- The ADAM method adjusts learning rates for each individual model parameter by approximating second order information about the objective function based on previously observed mini-batch gradients.
- Vanilla SGD refers to standard SGD without changes to the parameter updates.

SGD requires the choice of many optimization parameters. For example, learning rate determines the size of the parameter updates that SGD makes. Mini-batch size controls the size of the mini-batches used in a single parameter update. The momentum parameter determines the how much previous velocity vectors are dampened when using the momentum SGD variant. Choices of these optimization parameters have a large effect on the performance of SGD and its variants. See the dlTune action for automated ways of determining good optimization parameter values.

Parallelization of the SGD method can be performed in multiple ways: synchronous, elastic, and downpour. The total mini-batch size is calculated differently for each mode. For elastic and downpour modes, the total mini-batch size is **nThreads*miniBatchSize**. For synchronous mode, the total size is **nThreads*miniBatchSize*nWorkers**

Note: Large mini batches require a large learning rate. If the learning rate is not large, you might encounter model generalization issues.

Specifying the Optimizer Algorithm and Mode

The algorithm option that you specify determines which optimizer is used. There are four optimization options. The first three are SGD variants and the last one is L-BFGS:

- Vanilla (plain SGD)
- Momentum (SGD with momentum)
- ADAM (another popular SGD variant).
- L-BFGS (limited memory BFGS) on page 22.

The mode option determines how your chosen algorithm is parallelized across a grid. There are three different modes for parallelization:

- Synchronous Mode on page 21.
- Elastic Averaging Mode on page 21.
- Downpour Mode on page 21.

When running in SMP mode or with only one worker, synchronous mode is required. Also, L-BFGS supports only synchronous mode.

Synchronous Mode

The synchronous mode distributes gradient computations across worker nodes. Each worker computes the gradient for a portion of the data that resides on that worker, and the gradient is aggregated on the controller, which updates the parameters on the parameter server. The parameter server then sends the updated parameters to each worker, so the processing for the next iteration can proceed. Synchronous mode is the only available option for the L-BFGS algorithm.

Elastic Averaging Mode

Elastic Averaging is a stochastic gradient descent optimization that links the parameters of the concurrent processes of asynchronous stochastic gradient descent with an elastic force, such as a center variable stored by the parameter server. The elastic force allows the local variables to fluctuate farther from the center variable, which in theory allows for a fuller exploration of the parameter space. Empirical evidence indicates that this increased capacity for exploration leads to improved performance by finding new local optima.

The elastic averaging stochastic gradient descent method that SAS Deep learning uses is an asynchronous version of the Elastic Averaging SGD algorithm advanced by S. Zhang, A. Choromanska, and Y. LeCun. The elastic averaging SGD algorithm is motivated by the quadratic penalty method, but is re-interpreted as a parallelized extension of the averaging stochastic gradient descent algorithm. This algorithm provides fast convergent minimization while reducing the communication overhead between the master and local workers, while maintaining high quality performance as measured by the test error. Local workers do local updates to the parameters, and elastic averaging with the master (controller) asynchronously at syncFreq intervals of mini-batch iterations. Elastic optimization applies well to deep learning settings such as parallelized training of convolutional neural networks.

Downpour Mode

Downpour SGD is an asynchronous variant of the SGD method that was used by J. Dean et al. at Google. The downpour SGD algorithm runs multiple replicas of a model in parallel on subsets of the training data. Updates are sent to a parameter server, which is split across many machines. Workers compute gradients independently and do asynchronous communications to the controller. The controller updates the parameters and sends them back to the worker.

The SAS Deep Learning implementation of the downpour SGD method uses one parameter server and follows the algorithm defined in Large Scale Distributed Deep Networks by J. Dean et. al.

References

- “An Overview of Gradient Descent Optimization Algorithms”, Sebastian Ruder (2017), Insight Research Center – AYLIEN, https://arxiv.org/abs/1609.04747
- “Deep learning with Elastic Averaging SGD”, S. Zhang, A. Choromanska, and Y. LeCun (2015), Neural Information Processing Systems Conference (NIPS 2015), pp. 1–24. http://arxiv.org/abs/1412.6651
- “Large Scale Distributed Deep Networks”, J. Dean et. al. (2012), Google Inc., Mountain View, CA http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

Limited Memory BFGS Optimizer

Overview

Limited-memory BFGS (Broyden–Fletcher–Goldfarb–Shanno) is a popular quasi-Newton method used to solve large scale nonlinear optimization problems whose Hessian matrices are expensive to compute.

L-BFGS uses only the solutions and gradients from the most recent iterations to estimate the Hessian matrix. In most cases, the iteration history size can be very small, so the memory requirement dramatically is reduced. This makes L-BFGS very suitable for solving problems that have large numbers of variables.

Compared with SGD, L-BFGS requires more memory. It usually takes more epochs to reach the same level of the training or validation error. However, L-BFGS uses larger mini-batch sizes, which results in lowered computation cost per epoch. When the mini-batch size option is not specified, L-BFGS uses the entire epoch to compute the loss error and gradients.

Each step of L-BFGS attempts to approximate what the corresponding step of BFGS would do. Compared to BFGS, L-BFGS requires less memory and time for a single iteration. It can converge faster than BFGS because it can perform much more iterations within a given time span. However, compared to SGD, L-BFGS is more computational expensive and requires more memory, because it needs to approximate the Hessian matrix and stores search more search information.

L-BFGS Advantages

As an alternative to SGD optimizer, L-BFGS has its own advantages as follows:

- L-BFGS usually does not require extensive hyperparameter tuning.
- With the assistance of a linear search procedure, L-BFGS is usually more stable than SGD.
- L-BFGS can be naturally parallelized by computing the gradients in parallel.
- L-BFGS can handle large batch sizes well.

L-BFGS Implementation

The L-BFGS in SAS Deep Learning actions are implemented as follows:

- The L-BFGS line search method uses a new technique that uses log-linear convergence rates, which significantly reduces the average number of line search iterations.
- An improved line search strategy enables L-BFGS to make progress in bumpy regions.
- The mini-batch version of the current L-BFGS implementation is experimental.
- The L-BFGS implementation does not yet support GPU computing.

References

- “On the limited memory BFGS method for large scale optimization,” D.C. Liu and J. Nocedal. Mathematical Programming 45 (1989). pp. 503-528.
- “A comparison of algorithms for maximum entropy parameter estimation.” (PDF). Malouf, Robert (2002). Proc. Sixth Conf. on Natural Language Learning (CoNLL). pp. 49–55.

- "Scalable training of L_1-regularized log-linear models" (2007) G. Andrew and J. Gao, Proceedings of the 24th International Conference on Machine Learning, ISBN 9781595937933. DOI:10.1145/1273496.1273501.
- "Numerical Optimization: Understanding L-BFGS" (2 Dec 2014) A. Haghighi.

Hyperparameter Tuning

Overview

In machine learning, the optimization methods used to train models sometimes require you to choose parameter values that affect the performance of the optimizer. The parameters that you choose have an impact on the performance of the optimizer. That impact might be problem-specific. Hyperparameter tuning is a process that chooses the parameters for an optimization algorithm so that performance on a particular problem instance is improved. SAS Deep Learning tools support methods to automate the hyperparameter tuning process. We call this process *autotuning*. SAS Viya 3.3 and the SAS Deep Learning actions support autotuning of the hyperparameters for stochastic gradient descent (SGD).

The autotuning approach requires several steps:

1. Determine which hyperparameters you want to tune.
2. Determine reasonable ranges for the hyperparameter values.
3. Choose how many hyperparameter values you want to evaluate.
4. Assess the quality of the chosen hyperparameter values.

You must determine which hyperparameters you want to tune, and reasonable range values for those parameters before you can take advantage of the autotuning process. SAS uses internal derivative-free black box optimization algorithms that exploit rank and selection to choose the hyperparameters that you want to evaluate. For machine learning problems that use stochastic based optimization methods (SGD and variants), you can continue training using the tuned hyperparameters until the training process is complete. The quality of the model fit (usually expressed in terms of misclassification rate or error rate) is used for determining hyperparameter values.

The hyperparameter tuning process begins by choosing a number of hyperparameter sets in the ranges specified. Each hyperparameter set is trained for a small number of epochs, and bad performing hyperparameter sets are dropped from consideration. The remaining parameter sets are evaluated further until a specified number of tuning iterations is completed or until only a single hyperparameter set remains. By dropping bad performing hyperparameter sets quickly, good performing hyperparameter sets can be identified with less computational effort.

There are three options that drive the dlTune action. The maximum number of tuning iterations is specified by tuneIters. The proportion of hyperparameter sets retained between tuning iterations is denoted by the tuneRetention option (which must be between 0 and 1). Finally, the number of hyperparameter sets to evaluate is determined by the nParmSets option. Note that for each hyperparameter set evaluated, a copy of the model weights must be stored. For very large models, this might restrict the number of hyperparameter sets that can be evaluated.

During a tuning iteration, each remaining hyperparameter set is used to train a copy of the model weights for a small number of epochs. The maxIters (or maxEpochs) option controls the number of training epochs each hyperparameter set receives during one tuning iteration.

There are many tunable hyperparameters. To make it easy to choose hyperparameters to tune, the grammar for the dlTune action mimics the grammar for the dlTrain action. To add a hyperparameter to the list of tuned hyperparameters, just specify a lowerBound and upperBound instead of providing a value. For floating-point hyperparameter options, you may also specify the use of a logarithmic scale. A logarithmic scale is particularly useful for the learningRate parameter.

A complete list of tunable parameters follows:

Note: Not all parameters are applicable to all optimization methods.

- learningRate
- momentum
- gamma
- power
- regL1
- regL2
- dropout
- alpha
- damping
- beta1
- beta2
- gradientNoise
- miniBatchSize
- stepSize
- syncFreq

References

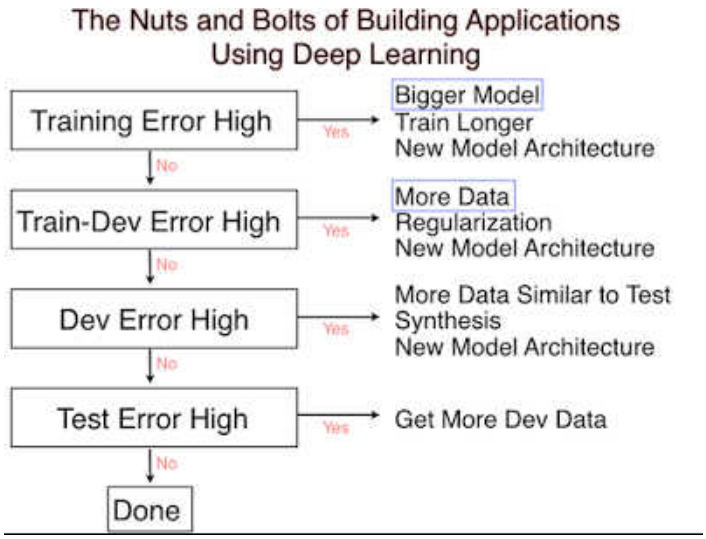
"Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization", L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, (2016), Cornell University Library, https://arxiv.org/abs/1603.06560.

Training Neural Networks

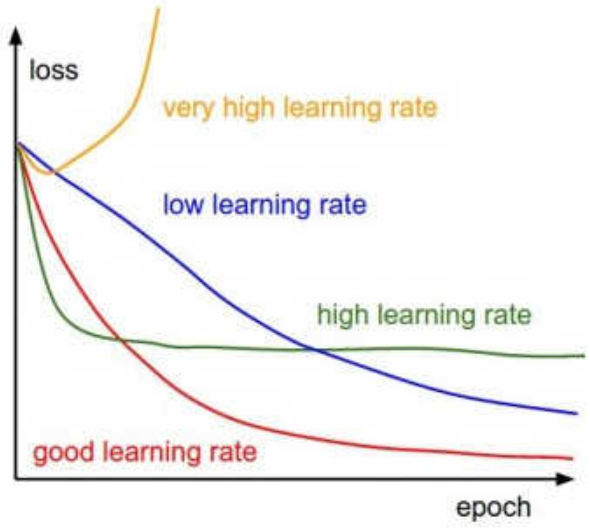
Training neural networks is a complex and challenging process, especially when the neural networks have many hidden layers, and each layer can have a large number of neurons. Deep neural networks and convolution neural networks are typical examples of networks that have multiple hidden layers. The following tips can help you train deep neural networks and convolution neural networks more efficiently.

- Flat loss (where the loss curve is flat and the loss cannot be improved) is very common when training deep neural networks. If you are experiencing significant flat loss, either the input data needs to be normalized, the weights need to be initialized more carefully, or some optimization parameters need to be tuned.
- The max batch size is the batch size per thread per worker. The actual batch size depends on data distribution. When data is not distributed evenly, as a result, some threads might not get enough data.
- Always randomly shuffle your data. The shuffle action in the table action set is recommended. A small block size can be applied when generating a new CAS table. Using small block sizes tends to distribute the data more evenly across threads and workers.
- Start with a simple and shallow neural network.
- Begin by using a small sample of training data to play with. Make sure you can overfit the small sample first.
- Dropout and L2 regularization techniques slow down the convergence speed.
- Small learning rate values are not unusual. The learning rate can be very small. 1.E-05 usually is the lower bound.

- Asynchronous stochastic gradient descent (SGD) algorithms are fast, because they enable a distributed architecture that is scalable and can use thousands of CPUs. However, it can be hard to train networks well using asynchronous SGD, even with good training data. You might try switching to synchronous SGD after you make a few asynchronous SGD runs.
- The reported fit error is the averaged results for all mini batches. The true fit error can be computed using dlScore on your training data.
- Pay attention to variances between the training error and the validation error. When the variance is large, there is a good chance that you over-fit the training data, or that your model is not generalized well. Try adding more layers or units into your model to see whether the difference between training error and validation error decreases.
- The SAS Deep Learning research and development team conducted tests using the CIFAR-10 and CIFAR-100 datasets, and observed that the exponential linear unit (ELU) activation function converged much faster than the rectified linear unit (ReLu) activation function.
- Nikolas Markou is an experienced machine learning practitioner and technical blogger. His Envision blog provides a very good list of Deep Learning Tips and Tricks.
- Slav Ivanov publishes a machine learning blog and offers 37 Reasons why your Neural Network is not working.
- Andrew Ng provides a useful chart correlating error rates and building deep learning applications:



- The following diagram shows the relationship between learning rate and loss:



Computational Considerations

The largest bottleneck to be aware of when constructing CNN architectures is the memory bottleneck. For instance, many GPUs have a limit of 6 or 12GB memory, with the best GPUs having about 12GB of memory. There are three major sources of memory consumption to keep track of:

- Watch your intermediate volume sizes. These are the raw number of activations (and gradients, of equal size) at every layer of the CNN. Normally, most of the activations are on the early layers of a CNN. The intermediate volumes are retained by the software because they are needed for back-propagation.
- Watch your parameter sizes: These are the numbers that hold the network parameters, their gradients during back-propagation, and commonly also a step cache in some optimization cases. Therefore, the memory to store the parameter vector alone must usually be multiplied by a factor of at least 3 or so.
- You must also watch your CNN model and how it uses miscellaneous memory, used for items such as image data batches, augmented versions, and so on.

After you have a rough estimate of the total number of values (for activations, gradients, and miscellaneous memory) in your CNN, the number should be converted to size in GB. Take the number of values, multiply by 4 to get the raw number of bytes (since every floating point is 4 bytes, or maybe by 8 for double precision), and then divide by 1024 multiple times to get the amount of memory in KB, MB, and finally GB. If your network does not fit, a common heuristic to “make it fit” is to decrease the batch size, since most of the memory is usually consumed by the activations.

Note: SAS Deep Learning actions internally use float to represent the model parameters, although they are stored as double precision in the CAS table.

Here are some computational considerations to remember when constructing RNN architectures using the SAS Deep Learning tools:

- SAS Deep Learning RNNs use all CPUs (or the number of threads specified) for both training and inference. Ideally, when data is evenly distributed and all sequences have the same length, RNN should be able to fully use all CPU resources. However, when sequences have variable lengths, the longest sequence in the mini-batch becomes the computational bottleneck. Padding does not count in sequence length because padding is discarded in RNN computations.
- The longest sequence becomes the bottleneck because there is sequential dependency during recurrent computation. Outliers that are much longer than other sequences can significantly slow down RNN tasks,

particularly training. It is recommended that you discard extremely long sequences if they are relatively rare, and strive to keep sequence lengths as close as possible.

- In addition to sequence length, other factors that decide computational cost with RNNs include token size (number of features per step/token), the number of neurons per layer, number of layers, and output vocabulary size (if applicable). Larger token size, larger number of neurons per layer, larger number of layers and larger output vocabulary size lead to more memory consumption and more CPU time usage. In particular, you should pay close attention to the output vocabulary size in text generation tasks. Output vocabulary size is the dominant factor for RNN memory consumption. A large output vocabulary can easily lead to out-of-memory errors and potentially cause the session to be killed.
- Although RNN computation can make full use of specified number of CPUs, it does not necessarily mean that users should always use maximum number of CPUs available. Using more CPUs usually leads to larger mini-batch size, and too large a mini-batch size can have a negative impact on convergence speed during training. Training time per epoch might be shorter with more CPUs, but if the mini-batch size is too large, the number of epochs needed to get a good model tends to increase. Model architects need consider the trade-off between number of CPUs, mini-batch size, and the required number of epochs for a good model.
- RNN architects can compensate by decreasing the mini-batch size per thread while increasing number of threads. However, you should remember that RNN computations are optimized in a manner that is more efficient when mini-batch size per thread is larger, up to a point. Eventually, the efficiency gain plateaus.
- Memory consumption by RNN computation increases linearly when the number of threads specified increases. So if you have a lot of CPUs, but limited memory, it is recommended that you not use all of your CPUs, in order to avoid running out of memory.

Ordering Model Weights

If you use other software to create a deep learning model that you want to import for use with SAS Deep Learning actions, this section provides information about creating a model weight table. The model weight table is a matrix that describes the layers and weights that are used in a deep learning network. You can import these weights to perform training (using the dlTrain action) or to perform scoring (using the dlScore action).

Model Weights Table

The SAS deep learning actions save and read weights from a weight table. It is laid out as follows:

LayerID
: A unique integer value to represent the layer number, which starts with 1. (The input layer is layer 0, but the input layer never has weights.)

WeightID
: An integer value to represent the weight assigned to a layer, which starts with 0 for each layer.

Weight
: Contains a single precision weight value. Only the first seven digits of the weight value are considered to be significant.

The following sections discuss how the weights associated with different types of layers are stored in the model.

Fully Connected Layer Weights

For a **fully connected layer**, the weights are laid out as follows:

1. Weight from first input to first neuron, weight from first input to second neuron, weight from first input to third neuron, and so on...
2. Weight from first input to last neuron, weight from second input to first neuron, weight from second input to second neuron, and so on...

3. Weight from second input to last neuron, weight from first input to first neuron, weight from last input to second neuron, and so on...
4. Weight from last input to last neuron, bias for first neuron, bias for second neuron, and so on... to bias for the last neuron.
5. If the setting for noBias=true, there are no bias weights.

Convolution Layer Weights

For a **convolution layer**, the weights are laid out as follows:

1. Weights for the first feature map, weights for the second feature map, weights for the third feature map... and so on.
2. Weights for the last feature map, bias for the first feature map, bias for the second feature map, bias for the third feature map ... and so on.
3. Weights for the last feature map, bias for the first feature map, bias for the second feature map, bias for the third feature map ... and so on.
4. If the setting for noBias=true, there are no bias weights.

Batch Normalization Layer Weights

For a **batch normalization** layer, the weights are laid out as follows:

If the source input to the batch normalization layer is structured as feature maps (such as the output of a convolution layer), there is one pair of gamma and beta bias weights for each feature map of the source layer.

If the source input to the batch normalization layer is *not* structured as feature maps (such as the output of a fully connected layer), there is one pair of gamma and beta bias weights for each neuron in the source layer.

Pooling Layer Weights

The **pooling layer** has no weights.

Residual Layer Weights

The **residual layer** has no weights.

Output Layer Weights

For an **output layer**, the weights are laid out in the same manner as the Fully Connected Layer Weights on page 27.

Recurrent Layer Weights

All weights and biases of the same recurrent layer have the same layer ID in the weight table. A bidirectional recurrent layer is implemented by adding two separate unidirectional recurrent layers, so the weights and biases of a bidirectional recurrent layer is stored in two separate unidirectional recurrent layers.

3

Importing Caffe Models into SAS Deep Learning

Overview of Caffe

What is Caffe?

Caffe is a deep learning framework created by Yangqing Jia while studying for his Ph.D at the University of California, Berkeley. The Berkeley Artificial Intelligence Research (BAIR) Lab continues to develop Caffe along with the help of the open-source community.

The Caffe framework is a BSD-licensed C++ library used to train and deploy deep learning models on general-purpose or custom hardware architectures. Plaintext schemas define deep learning models while simple text files control model optimization. Caffe provides a command-line interface, but exposes both Python and Matlab interfaces as well.

Components of a Caffe Model

A Caffe model consists of at least three files:

- **Deep learning network file:** The network file defines the deep learning architecture along with key architecture parameters. This file has a `.prototxt` extension and is required to train or deploy a model.
- **Solver definition file:** The solver file coordinates model optimization. This file has an extension of `.prototxt` and is required for training a model.
- **Model parameter file:** The model parameter file stores model parameter settings that were determined during the training process. Model parameter files have `.caffemodel` extensions if in BINARYPROTO format, or `.caffemodel.h5` extensions if in HDF5 format.

Where Can I Go for More Information?

Online resources are available through the University of California, Berkeley and the Berkeley Artificial Intelligence Labs to help understand basic components of Caffe and Caffe models, as well as a basic approach to the following:

- The following overview and general tour of Caffe is provided by BAIR and UC-Berkeley: Caffe Tutorial. The practical guide presents the Caffe principles, as well as documenting the following Caffe topics:
  - Nets, Layers, and Blobs: the anatomy of a Caffe model.
  - Forward / Backward: the essential computations of layered compositional models.
  - Loss: the task to be learned is defined by the loss.
  - Solver: the solver coordinates model optimization.
  - Layer Catalogue: the layer is the fundamental unit of modeling and computation. Caffe's catalogue includes layers for state-of-the-art models.
  - Interfaces: command line, Python, and MATLAB Caffe.
  - Data: how to structure data for model input.
- This Caffe tutorial by Evan Shelhamer, Jeff Donahue, and Jon Long provides an introduction to deep learning and convolutional networks, and also performs a hands-on overview of the Caffe framework. DIY Deep Learning for Vision: A Full-Day Caffe Tutorial
- See Caffe models in action! Interact with the Caffe Web Image Classification Demo.

  The Caffe demo operates as follows: From the linked classification model demo site on your browser, use the link space to provide the URL of an image on the web, or upload a locally saved image file to the Caffe demo library for classification. Easy screen controls are provided.

  For example, this is how the image **`gatorsbadgersncaa.jpg`**, from Sports Illustrated's SEC Season in Review (2017) is analyzed by the Caffe Web Image Classification demo:

References

- Caffe Tutorial, Berkeley Artificial Intelligence Research Lab, University of California, Berkeley.

Obtain and Configure Caffe Environment

Preparation

Before you can use Caffe models, you need to install and build your own Caffe environment. Go to this GitHub page and select the green **Clone or Download** button to secure the required software installer.

When you clone or download Caffe, this documentation refers to the initial directory that you use as the `$CAFFE_ROOT` directory. `$CAFFE_ROOT` is a representative placeholder for name of the file directory that you choose as the root for your Caffe installation. It's a good idea to make a note about the location of your Caffe root directory when you create it, because you will need to reference it again.

Prerequisites

Caffe has several dependencies:

- CUDA is required for GPU mode. CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).
- Basic Linear Algebra Subprograms, called BLAS, is required. You can obtain a BLAS library via ATLAS, or Intel's Math Kernel Library MKL, or from the optimized BLAS library OpenBLAS .
- Portable C++ source libraries from Boost.org, where the Boost library is version 1.55 or later. Boost offers version 1.65 of the source library here.

The following are optional dependencies that you might encounter:

- If you use OpenCV, you will need the Open-Source Computer Vision Library. OpenCV is released under a BSD license, which supports its free use for academic and commercial use. OpenCV has C++, C, Python, and Java interface, and supports Windows, Linux, Mac OS, iOS, and Android. You should use a version of OpenCV later than OpenCV 2.4. You can download OpenCV 3.3 here.
- You might need one of the LMDB or LevelDB I/O libraries. LMDB is a Lightning Memory-Mapped Database library, and LevelDB is a fast key-value storage library that provides ordered mappings from string keys to string values.

  Note: LevelDB requires Snappy. Snappy is a compression/decompression library. You can find more information about Snappy here.

- For GPU acceleration, you might need the NVIDIA CUDA Deep Neural Network library CuDNN, version 6 or later. To speed up your Caffe models, install cuDNN, then uncomment the `USE_CUDNN := 1` flag in `Makefile.config` when installing Caffe. Acceleration is automatic. NVIDIA's cuDNN is available from NVIDIA's Accelerated Computing website here

  If you do not plan on using GPU acceleration, using CPU only, uncomment the `CPU_ONLY := 1` flag in `Makefile.config` to configure and build Caffe without CUDA. This is helpful for cloud or cluster deployment.

- If you are using Python Caffe, you might need Python 2.7 or Python 3.3+. You can download from Python here. You might also need NumPy version 2.7 or later. Latest versions of NumPy are available here.

Install and Configure Your Caffe Environment

After your clone downloads, you need to install your Caffe environment.

Follow the step-by-step instructions on Berkeley's Caffe installation page to install the environment on your operating system. BAIR recommends you read the installation guide thoroughly before beginning, and note the details that are particular to your platform. BAIR supports Caffe on Ubuntu 16.04–12.04, OS X 10.11–10.8, and through Docker and AWS.

To prepare your Caffe model files for use with some compilers, you might need to edit your `Makefile.config` file. The `CXXFLAGS` parameter in `Makefile.config` is defined as `CXXFLAGS += -MMD -MP`. It might be necessary to change the `CXXFLAGS` parameter definition to `CXXFLAGS += -MMD -MP -std=c++11`.

Obtain HDF5 Conversion Tool Code

Caffe model parameter files are saved in one of two file formats. The first format, called BINARYPROTO, is a binary serialization of the Google protocol buffer format. The BINARYPROTO format has a `.caffemodel` file extension. The second format is called HDF5. HDF5 is a file format designed for storing and managing data. The HDF5 format has a `.caffemodel.h5` file extension.

If your model parameter file is in BINARYPROTO format, you must convert your model parameter file to HDF5 format before you can import it for use with the SAS Deep Learning tools. You use the `convertToHdf5` executable conversion tool to convert your Caffe model parameter files from BINARYPROTO to HDF5 format. The HDF5 model parameter files can then be imported to populate a deep neural network.

Obtain the conversion tool source code, `convertToHdf5.cpp`. The zipped convertToHdf5 conversion tool is available from the SAS support page for Deep Learning Models and Utilities.

Download and expand the ZIP file, and then copy the file `convertToHdf5.cpp` to the corresponding path on your platform: `$CAFFE_ROOT/tools/`.

Note:

`$CAFFE_ROOT` represents the file directory that is the root for your Caffe installation, as discussed in Preparation on page 31.

Compile Caffe

When you download or update Caffe, you must compile Caffe before use. Caffe can be compiled with either Gnu's Make or the open-source CMake. Gnu's Make is officially supported, while CMake is supported by the community.

If you want to compile with Gnu's Make, you can configure the build by copying and modifying the example `Makefile.config` for your setup. The defaults should work, but you should uncomment the relevant lines if you are using Anaconda Python. If desired, you can run `make distribute` to create a `distribute` directory

that will have all the Caffe headers, compiled libraries, binaries, and so on, that are needed for distribution to other machines.

```
cp Makefile.config.example Makefile.config
# Adjust Makefile.config (for example, if using Anaconda Python, or if cuDNN is desired)
make all
make test
make runtest
```

- For GPU accelerated Caffe, no changes are needed.
- For cuDNN acceleration using NVIDIA's proprietary cuDNN software, uncomment the `USE_CUDNN := 1` switch in Makefile.config. cuDNN is sometimes but not always faster than Caffe's GPU acceleration.
- For CPU-only Caffe, uncomment `CPU_ONLY := 1` in `Makefile.config`.

Other users prefer to compile Caffe using CMake. In lieu of manually editing `Makefile.config` to configure the build, Caffe offers an unofficial CMake build, thanks to members of the Caffe community. It requires CMake version 2.8.7 or later. The basic steps are as follows:

```
mkdir build
cd build
cmake ..
make all
make install
make runtest
```

For more options and details about CMake scripts, see GitHub: Improved CMake Scripts #1667.

References

- "Installing Caffe", Berkeley Artificial Intelligence Research Lab, University of California, Berkeley, http://caffe.berkeleyvision.org/installation.html
- Atlas open-source charts and map utility, by Sourceforge. http://atlas.sourceforge.net/
- NVIDIA CUDA DNN deep neural networking library, NVIDIA Corporation. https://developer.nvidia.com/cudnn
- CUDA and NVIDIA GeForce Technology, NVIDIA Corporation, https://www.geforce.com/hardware/technology/cuda
- Math Kernel Library and Basic Linear Algebra Systems, Intel Corporation, https://software.intel.com/en-us/mkl/documentation/get-started
- Intel OpenBLAS, Intel Corporation. https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines
- C++ libraries from Boost.org. http://www.boost.org/
- Python Caffe, https://www.python.org/downloads/

Download a Deep Learning Model

Now you can download a Caffe model. The Caffe Model Zoo is an excellent starting point and provides links to models for a variety of applications. On the Caffe Model Zoo Wiki page, navigate to the Table of Contents, and then click on any of the supplied model links. This takes you to a section devoted to that particular model. From that section, you will be able to download model files. Typically, these models are in BINARYPROTO format, and must be converted to HDF5 format before use with SAS Deep Learning actions. You can also obtain pre-converted models from the SAS Institute support site page for Visual Data Mining and Machine Learning.

Downloaded deep learning models are subject to usage conditions that are established by the respective authors. Please pay attention to requirements for citation and license for any downloaded model.

Caffe Model Zoo

To facilitate the sharing of models, the Caffe Model Zoo uses the model zoo framework. This framework provides the following:

- a standard format for packaging model information.
- tools to upload and download model information from GitHub gists, and to download `.caffemodel` or `caffemodel.h5` binaries. GitHub Gists are a great way to share your work. You can share single files or full applications using gists. Every gist is a Git repository, which means that it can be forked or cloned.
- a central Wiki page for sharing model info Gists.

SAS Support Site

You can obtain Caffe models converted for use with SAS Deep Learning actions from the SAS Institute support site for Deep Learning Models and Tools.

This support page contains deep learning models, tools, and examples. The Deep Learning Models section contains popular convolutional neural models that have been converted for use with SAS Deep Learning actions. The Deep Learning Examples section contains support files for two narrated Caffe model examples. The Deep Learning Utilities section contains handy tools, such as the conversion utility for use with SAS Deep Learning actions, as well as a library of SAS and Python code snippets and function definitions that can be used to simplify the definition of SAS Deep Learning models.

The converted models that are provided are popular convolutional neural networks used to perform image classification. The collection of converted models will grow over time to encompass other application domains.

Convert Downloaded Model from BINARYPROTO Format to HDF5 Format

If you download a model in BINARYPROTO format, it must be converted to HDF5 format for use with SAS Deep Learning actions. Models downloaded in HDF5 format may be used without modification by the SAS Deep Learning actions.

HDF5 Model Conversion Tool

You use the HDF5 conversion tool to change your Caffe model parameter files from BINARYPROTO format to HDF5 format. The HDF5 conversion tool is an executable Linux command-line program called **convertToHdf5**.

The HDF5 conversion tool is distributed as an uncompiled C++ file named `convertToHdf5.cpp`. You can download the uncompiled source code for the HDF5 model conversion tool from the SAS support site for Deep Learning Models and Tools.

The `convertToHdf5` executable file is generated when you add the C++ source code file `convertToHdf5.cpp` to your local Caffe environment, and then use your favorite C++ tools to compile the Caffe environment.

The following section discusses how to invoke the compiled HDF5 conversion program.

Type 1 and Type 2 HDF5 Format Conversions

You can run the HDF5 conversion program in two ways: type 1 and type 2 conversions. The difference between the two is that a type 1 conversion does not need a mapping file, but a type 2 conversion does. A layer mapping file is required when importing a Caffe model that uses a batch normalization layer followed by a scaling layer, due to implementation differences between the SAS Deep Learning actions and Caffe. For such a model, the SAS naming convention is that the scale layer must be a single-name concatenation of <*batch-normalization-layer-name*> and <*_scale*>. The mapping file allows the conversion program to rename the necessary layers to conform to the SAS naming convention while writing the HDF5 format model parameter file.

The simplest conversion is a Type 1 conversion. The Type 1 invocation syntax is as follows:

`convertToHdf5<`*`Caffe-network-definition-file`*`.prototxt><`*`Caffe-model-paremeter-file`*`.caffemodel><`*`name-for-HDF5—converted-Caffe-model`*`.caffemodel.h5>.`

Here is an example of a Type 1 `convertToHdf5` invocation:

`convertToHdf5 VGG_ILSVRC_16_layers.prototxt VGG_ILSVRC_16_layers.caffemodel VGG_ILSVRC_16_layers.caffemodel.h5`, where

VGG_ILSVRC_16_layers.prototxt
is the network definition file for a VGG-16 model. VGG-16 is one of the CNN architectures used by the VGG team in the ILSVRC-2014 competition. Details about the network architecture can be found in the white paper by Karen Simonyan and Andrew Zisserman on Very Deep Convolutional Networks for Large-Scale Image Recognition.

VGG_ILSVRC_16_layers.caffemodel
is the VGG-16 model parameter file in BINARYPROTO format.

VGG_ILSVRC_16_layers.caffemodel.h5
is the desired output name for your converted VGG-16 model parameter file in HDF5 format.

Type 2 conversions require you to specify an extra input parameter: the required layer mapping file. The type 2 invocation syntax is as follows:

`convertToHdf5 <`*`Caffe-network-definition-file.prototxt`*`><`*`layer-mapping-file.txt`*`><`*`Caffe-model-paremeter-file.caffemodel`*`><`*`name-for-HDF5—converted-Caffe-model.caffemodel.h5`*`>.`

Here is an example that shows a Type 2 invocation of `convertToHdf5` to convert a ResNet-50 Caffe model from BINARYPROTO format to HDF5 format.

`convertToHdf5 ResNet-50-deploy.prototxt ResNet-50-mapping.txt ResNet-50-model.caffemodel ResNet-50-model.caffemodel.h5`, where

ResNet-50-deploy.prototxt
is the Caffe network definition file for ResNet-50. ResNet-50 is one of the CNN architectures developed by a team from Microsoft that won the ImageNet Large Scale Visual Recognition Competition (ILSRVC) in 2015.

ResNet-50-mapping.txt
is the text file that you use to provide the original ResNet-50 layers with new names. Each line of the `ResNet-50-mapping.txt` file should have the form of *original_layer_name new_layer_name*, where the value for *original_layer_name* is taken from a layer defined in `ResNet-50-deploy.prototxt`, and the *new_layer_name* is the corresponding new layer name that you supply.

ResNet-50-model.caffemodel
is the name of the ResNet-50 model parameter file in BINARYPROTO format.

ResNet-50-model.caffemodel.h5
is the name for the output ResNet-50 model parameter file that is converted to HDF5 format.

Import Model into SAS Deep Learning

Order of Operations

The final step is to import your Caffe model for use with SAS Deep Learning actions.

Here is an outline of the process. In this outline, we use the batch normalization version of a LeNet model as an example.

1 Download the archive of your model files. For illustration purposes, download convert_lenet_example.zip from the Deep Learning Models and Tools page on the SAS support site. When expanded, you find the following files:

   - network definition file `lenet_train_test_bn.prototxt`
   - mapping file `lenet_mapping.txt`
   - model parameter file `lenet.caffemodel`

2 Examine the network definition file `lenet_train_test_bn.prototxt`, and notice the mappings that are defined in the mapping file `lenet_mapping.txt`. Recall the convention used when importing a Caffe model for use with SAS: the scale layer must be a single-name concatenation of <*batch-normalization-layer-name*> and <*_scale*>. For example, if the batch normalization layer was called `batch_norm`, then the naming convention requires the scale layer to be called `batch_norm_scale`.

   Note: Since models from the Caffe Model Zoo are not expected to follow the naming conventions that are required to import the models into SAS, you are responsible for providing a mapping file that meets the SAS layer name requirements.

3 Convert the Caffe model into HDF5 format, using the type 2 on page 35 invocation of the `convertToHdf5` executable. For information about obtaining the `convertToHdf5` executable, see HDF5 Model Conversion Tool on page 34.

4 Define a SAS Viya LeNet deep learning model. If you are using a Python client, you could write a Python function to define the LeNet model that resembles the following:

```
# LeNet model definition (batch normalization version)
def LeNet_Model(s):

   # instantiate model
   s.buildModel(model=dict(name='LeNet',replace=True),type='CNN')

   # add input layer mnist
   s.addLayer(model='LeNet', name='mnist',
              layer=dict( type='input', nchannels=1, width=28, height=28,
              scale=0.00392156862745098039))

   # add convolution layer: 5*5*20
   s.addLayer(model='LeNet', name='conv1',
              layer=dict( type='convolution', nFilters=20, width=5, height=5,
                          stride=1, act='identity', noBias=True, init='xavier'),
              srcLayers=['mnist'])

   # add batch normalization layer conv1_bn:
   s.addLayer(model='LeNet', name='conv1_bn',
              layer=dict( type='batchnorm', act='relu'), srcLayers=['conv1'])
```

```
# add pooling layer pool1 2*2*2
s.addLayer(model='LeNet', name='pool1',
           layer=dict(type='pooling',width=2, height=2, stride=2, pool='max'),
           srcLayers=['conv1_bn'])

# add convolution layer conv2: 5*5*50
s.addLayer(model='LeNet', name='conv2',
           layer=dict( type='convolution', nFilters=50, width=5, height=5,
                       stride=1, act='identity', noBias=True, init='xavier'),
           srcLayers=['pool1'])

# add batch normalization layer conv2_bn
s.addLayer(model='LeNet', name='conv2_bn',
           layer=dict( type='batchnorm', act='relu'), srcLayers=['conv2'])

# add pooling layer pool2 2*2*2
s.addLayer(model='LeNet', name='pool2',
           layer=dict(type='pooling',width=2, height=2, stride=2, pool='max'),
           srcLayers=['conv2_bn'])

# add fully connected layer ip1
s.addLayer(model='LeNet', name='ip1',
           layer=dict(type='fullconnect',n=500, init='xavier', act='relu'),
           srcLayers=['pool2'])

# add output layer ip2
s.addLayer(model='LeNet', name='ip2',
           layer=dict(type='output',n=10, init='xavier', act='softmax'),
           srcLayers=['ip1'])
```

5 Import the model parameters using the **DlImportModelWeights** action from SAS Viya. An example action invocation using Python resembles the following:

```
# import model weights
s.dlimportmodelweights(model='LeNet',
                       modelWeights={"name":"tkdlImportedWeights",
                                     "replace":True},
                       formatType="CAFFE",
                       weightFilePath=fullyQualifiedModelPath)
```

6 Your trained model is now imported for use with the SAS Deep Learning actions.

For an example of using a converted model, see Classifying ImageNet Data with an Imported Caffe Model on page 37 in the following section.

Classifying ImageNet Data with an Imported Caffe Model

Prerequisites for this Exercise

This exercise provides an example of importing a Caffe model and evaluating the performance of the model on a subset of the ImageNet 2012 data. To utilize the code presented in this example, you must first do the following:

- Create a directory named $IMAGENET_EXAMPLE in your local computing environment. This directory is for your ImageNet example files.

- Download the archived model definition file library models.zip from the Deep Learning Model Tools and Utilities page on the SAS support web site, and unpack the file contents to your directory $IMAGENET_EXAMPLE.
- Download the sample program imagenet_example.zip from the Deep Learning Model Tools and Utilities page on the SAS support web site, and unpack the file contents to $IMAGENET_EXAMPLE. The file contents include a README text file, a shell script (.sh) to make a file list, as well as example SAS and Jupyter notebook code files.
- Create a directory named $MODEL_ROOT in your local computing environment. This directory is for your converted Caffe model parameter files.
- Download the archived converted model parameter file that you want to evaluate from the model parameter file directory on the SAS support web site. For this example, we will use the VGG-16 model, so you would download vgg16.zip and unpack the contents to your $MODEL_ROOT directory.
- Using your site's SAS Viya software, start a SAS CAS session. Take note of the port number associated with your CAS server connection, because you will use it later in some of your code.

Download Image URLs for Input Data

First, we download the URLs for images representing the class "ladybug" from ImageNet. ImageNet is an ongoing research effort by the Stanford Vision Lab to provide researchers around the world an easily accessible image database. The image database is organized according to the WordNet hierarchy, in which each node of the hierarchy is depicted by hundreds and thousands of images.

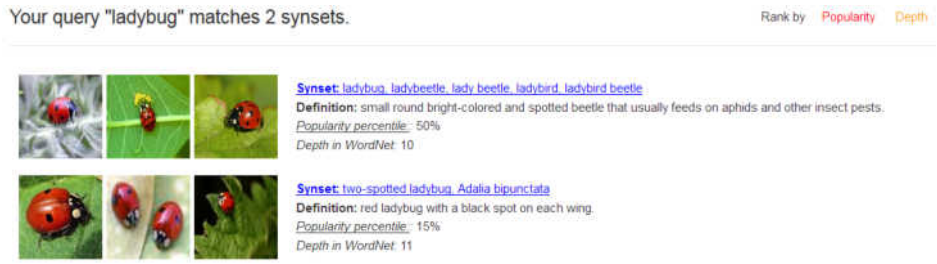
1 Choose one of the image classes included in the ImageNet classification challenge. This example uses the *ladybug* class. You can find Imagenet's 2012 synsets for object classification here:

   http://image-net.org/challenges/LSVRC/2012/browse-synsets.

   Note: You can perform this example using a different image class of your own choice, but the reference calculations displayed in the final example step are values for the ladybug image class. If you use a different image class, the image misclassification rates that you compute will not match the misclassification rates shown for the ladybug image class in this example.

2 Go to the ImageNet URL download page and type `ladybug` in the Search box.

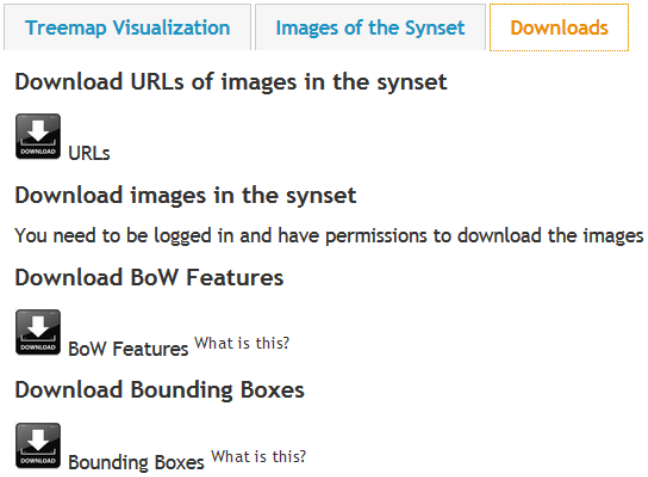
3 Your search should return two hits.



4 Click the link that reads Synset: ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle

   This opens the corresponding synset (image class) page with three tabs: **Treemap Visualization**, **Images of the Synset**, and **Downloads**.

5 Click the **Downloads** tab.

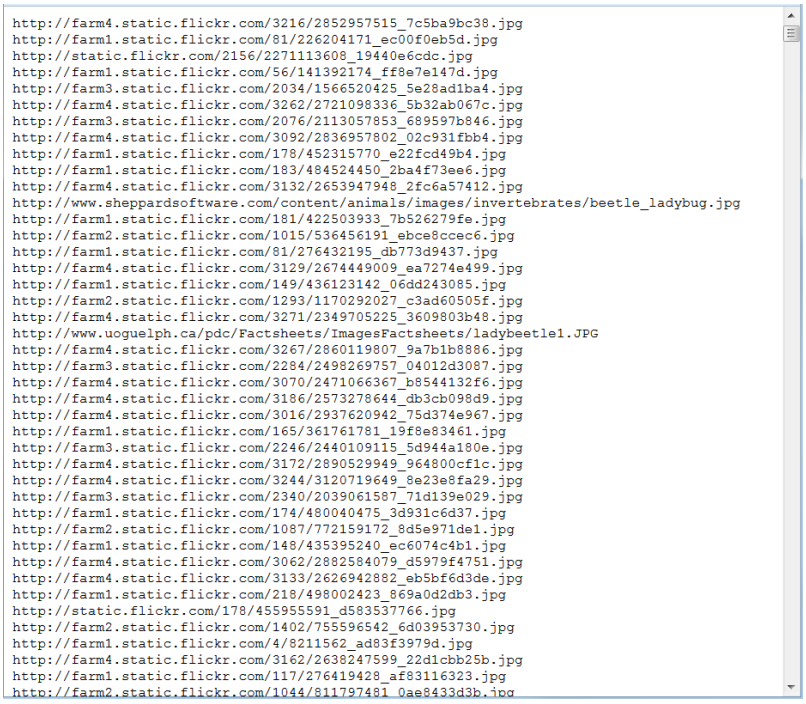


Click the icon to Download URLs of Images in the synset.

Save Downloaded Image URLs as Text File

After you click the icon to Download URLs of Images in the synset, do the following:

1 Your browser window will populate with a large list of image URLs associated with the synset that resembles the following:



Select and copy the contents of the URL list in your browser window.

2 Open the text editor of your choice, paste in the list of URLs you copied from Imagenet, and save the file in text format: `ladybug_files.txt`.

Create a Directory to Hold Images

1 Next, you need a file structure that can hold your downloaded images and model files. Call this directory `$IMAGE_ROOT/ladybug`, where IMAGE_ROOT is an environmental variable that represents the base path for your images directory.

2 Copy `ladybug_files.txt` to `$IMAGE_ROOT`.

3 Navigate to `$IMAGENET_EXAMPLE` and run the shell script `make_file_list.sh` that you previously unpacked. Use the following syntax:

```
./make_file_list.sh $IMAGE_ROOT ladybug_files.txt ladybug
```

The `make_file_list.sh` script downloads the images that correspond to the links in your URL list. Broken or defective URLs on the list are ignored.

Note: The download process may take some time, as there are approximately 2000 images in the ladybug synset. After the shell script filters out the broken and defective image links, you will obtain more than 1500 images.

Open, Modify, and Run ImageNet Model in Jupyter Notebook

1 Start your Jupyter notebook from the $IMAGENET_EXAMPLE directory. If you need information about how to download and start a Jupyter notebook, Unidata provides a useful online guide: How to Start and Run a Jupyter Notebook.

2 Locate the file `score_imagenet_example.jpynb`, and open it. You need to define model variables by editing the Jupyter notebook cell that contains the comment `# define variables used to control test`. Modify the variables according to your deployment scenario. The examples below show formatted code, but you will need to provide the parameters for your site, such as server name, port number, image and model file paths, and so on.

Here is a list of the variables you need to define in the `# define variables...` cell before you can run the notebook, and the type of information that you need to supply for each variable:

casHost
: "<*your-server-name*>" Example: `casHost = "dlserver001.unx.mycompany.com"`

casPort
: <*unique-port-number*> Example invocation: `casPort=01812`

imagePath
: <*path-to-images*> Example invocation: `imagePath="/home/deeplrn/data/ladybug"`

modelPath
: <*path-to-Caffe-model-files*> Example invocation:`modelPath="/home/deeplrn/imported_models"`

modelName
: <*model-name*> Example invocation:`modelName="VGG16"`

3 Run all of the cells in your notebook and examine your results. For reference, a VGG-16 model gives top 1 and top 5 misclassification error rates of 15.3% and 4.2 % respectively.

Deep Learning Action Set: Examples

Example 1: Add a Layer to a Deep Learning Model

You can use CASL to add a layer to a deep learning model using the addLayer action.

The following example shows how you can use CASL to add a layer to a deep learning model using the addLayer action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";      /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                              /* 2 */
   session casauto;

   <Additional steps before adding a layer>            /* 3 */

   deepLearn.addLayer /                                /* 4 */
      layer={type="INPUT"}
      modelTable={name="Model_table_name"}
      name="Layer_name";
   run;

quit;                                                  /* 5 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Before you add a layer to your model, make sure that you have first built a model. For more information about building a model, see Build a Deep Learning Model on page 45.

4 Calls the addLayer action in the Deep Learning action set using the layer type, model table, and layer name that you want to use.

Note: You need to modify the `name` value for the `modelTable` parameter to fit the model table that you want to use. You also need to modify the `name` parameter to specify the layer name that you want to use. In this example, the layer type is specified as “INPUT”. If you want to use a different type of layer, you need to modify the value of the layer `type` parameter.

5 Ends the code to run a CAS action in SAS.

The following code builds an empty recurrent neural network model using the model table mtRnn, and then adds input, recurrent, and output layers to it:

```
proc cas;
```

```
   session casauto;

   deepLearn.buildModel /
      modelTable={name="mtRnn",
                  replace=TRUE
                 }
      type="RNN";
   run;

   deepLearn.addLayer /
      layer={type="INPUT"}
      modelTable={name="mtRnn"}
      name="data";
   run;

   deepLearn.addLayer /
      layer={type="recurrent"
             n=50
             act='sigmoid'
             init='xavier'
             rnnType='gru'
            }
      modelTable={name="mtRnn"}
      name="rnn1"
      srcLayers={"data"};
   run;

   deepLearn.addLayer /
      layer={type="output"
             act='softmax'
             init='xavier'
            }
      modelTable={name="mtRnn"}
      name="outlayer"
      srcLayers={"rnn1"};
   run;

quit;
```

Here are the results from building an empty recurrent neural network model:

The SAS System

Results from deepLearn.buildModel

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS( )	mtrnn	1	5

Here are the results from adding an input layer to the recurrent neural network model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS( )	mtrnn	10	5

Here are the results from adding a recurrent layer to the recurrent neural network model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS( )	mtrnn	23	5

Here are the results from adding an output layer to the recurrent neural network model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS( )	mtrnn	35	5

Example 2: Build a Deep Learning Model

You can use CASL to build a deep learning model using the buildModel action.

The following example shows how you can use CASL to build a deep learning model using the buildModel action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                             /* 2 */
   session casauto;

   <Additional steps before building a model>         /* 3 */

   deepLearn.buildModel /                             /* 4 */
      modelTable={name="Model_table_name"}
      type="CNN";
   run;

quit;                                                 /* 5 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, you might want to load sample data for your model. For more information about loading a sample data source, see Load a Sample Data Source for a Deep Learning Model on page 66.

4 Calls the buildModel action in the Deep Learning action set using the model table and the model type that you specify.

Note: You need to modify the `name` value for the `modelTable` parameter to fit the model table name that you want to use. In this example, the model type was specified as a convolutional neural network (CNN). If you want to build a different type of model, you need to modify the value of the `type` parameter to specify the model type that you want to build.

5 Ends the code to run a CAS action in SAS.

The following code builds an empty convolutional neural network (CNN) using the model table name mTable:

```
proc cas;

   session casauto;

   deepLearn.buildModel /
      modelTable={name="mTable"}
      type="CNN";
   run;

quit;
```

Here are the results from building a model:

The SAS System

Results from deepLearn.buildModel

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS( )	mtable	1	5

Example 3: Export a Deep Learning Model

You can use CASL to export a deep learning model using the dlExportModel action.

The following example shows how you can use CASL to export a deep learning model using the dlExportModel action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";          /*1*/
cas casauto;
caslib _all_ assign;

proc cas;                                                  /*2*/
```

```
   session casauto;

   <Additional steps before exporting a model table>       /*3*/

   deepLearn.dlExportModel /                               /*4*/
      casout={name="Out_table_name"}
      initWeights={name="IW_table_name"}
      modelTable={name="Model_table_name"};
   run;

quit;                                                      /*5*/
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, make sure that you have trained your model before you export it. For more information about training a model, see Train a Deep Learning Model on page 56.

4 Calls the dlExportModel action in the Deep Learning action set using the output table that you specified to store the generated model, the in-memory table that contains the model weights, and the in-memory table that is the model.

   Note: You need to modify the `name` values for the `casout`, `initWeights`, and `modelTable` parameters to the fit the output table, the model weights in-memory table, and the in-memory model table that you want to use.

5 Ends the code to run a CAS action in SAS.

To illustrate the possible steps that you might need to take before exporting a deep learning model, consider the following example. In it, a library is added, and a table is loaded with sample data. Next, an empty recurrent neural network model is created, and layers are added to it.

```
proc cas;

   session casauto;

   table.addCaslib /
      datasource={srcType="PATH"}
      name="My_library_name"
      path="/u/userDir/directory_with_sample_data";
   run;

   table.loadTable result=r /
      caslib="My_library_name"
      importOptions={fileType="CSV"}
      path="My_file_name"
      casOut={name="CAS_file_name", replace=true};
   run;

   deepLearn.buildModel /
      modelTable={name="mtRnn",
                  replace=TRUE
                 }
      type="RNN";
   run;
```

```
   deepLearn.addLayer /
      layer={type="INPUT"}
      modelTable={name="mtRnn"}
      name="data";
   run;

   deepLearn.addLayer /
      layer={type="recurrent"
             n=50
             act='sigmoid'
             init='xavier'
             rnnType='gru'
            }
      modelTable={name="mtRnn"}
      name="rnn1"
      srcLayers={"data"};
   run;
```

After layers have been added to the model, training is performed using the model and the input data. Finally, the model is exported using the model weights that have been created and the model table. The actual steps that you should take before exporting a deep learning model varies depending on your modeling goals. For an example with sample data that you can run, see Build a Deep Learning Model with Tabular Data on page 68.

```
   deepLearn.addLayer /
      layer={type="output"
             act='softmax'
             init='xavier'
            }
      modelTable={name="mtRnn"}
      name="outlayer"
      srcLayers={"rnn1"};
   run;

   deepLearn.dlTrain /
      inputs={{name="Variable_name"}}
      modelTable={name="mtRnn"}
      modelWeights={name="MW_table_name"}
      table={name="CAS_file_name"};
   run;

    deepLearn.dlExportModel /
      casout={name="Out_table_name"}
      initWeights={name="MW_table_name"}
      modelTable={name="mtRnn"};
   run;

quit;
```

Example 4: Import Model Weights

You can use CASL to import model weights from an external source using the dlImportModelWeights action.

The following example shows how you can use CASL to import model weights from an external source using the dlImportModelWeights action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";          /*1*/
cas casauto;
caslib _all_ assign;

proc cas;                                                  /*2*/
   session casauto;

   <Additional steps before importing model weights>       /*3*/

   deepLearn.dlImportModelWeights /                        /*4*/
      modelTable={name="Model_table_name"}
      modelWeights={name="MW_table_name"}
      weightFilePath="Model_Weight_Path_And_File";
   run;

quit;                                                      /*5*/
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, make sure that you have built a model before you import model weights. For more information about building a model, see Build a Deep Learning Model on page 45.

4 Calls the dlImportModelWeights action in the Deep Learning action set using the in-memory table that is the model, the in-memory table that is used to store the model weights, and the fully qualified path and filename for the external model weight file.

   Note: You need to modify the `name` values for the `modelTable` parameter and the `modelWeights` parameter to specify the in-memory model table that you want to use and the in-memory table that is used to store the model weights. You also need to modify the value for the `weightFilePath` parameter to specify the fully qualified path and filename for the external model weight file.

5 Ends the code to run a CAS action in SAS.

For example, consider that you want to conduct a deep learning analysis with a preexisting model weights file. The following example creates an empty convolutional neural network, and then adds layers to it:

```
proc cas;
   session casauto;

   deepLearn.buildModel /
      modelTable={name="LeNet",
                  replace=TRUE
                 }
      type="CNN";
   run;

   deepLearn.addLayer /
      layer={type="INPUT"
             nchannels=1
             width=28
             height=28
```

```
           scale=0.00392156862745098039
          }
   modelTable={name="LeNet"}
   name="mnist";
run;

deepLearn.addLayer /
   layer={type="CONVO"
          nFilters=20
          width=5
          height=5
          stride=1
          init="XAVIER"
          act="IDENTITY"
         }
   modelTable={name="LeNet"}
   name="conv1"
   srcLayers={"mnist"};
run;

deepLearn.addLayer /
   layer={type="POOL"
          width=2
          height=2
          stride=2
          pool='max'
         }
   modelTable={name="LeNet"}
   name="pool1"
   srcLayers={"conv1"};
run;
```

The following adds additional layers to the convolutional neural network model:

```
deepLearn.addLayer /
   layer={type="CONVO"
          nFilters=50
          width=5
          height=5
          stride=1
          init="XAVIER"
          act="IDENTITY"
         }
   modelTable={name="LeNet"}
   name="conv2"
   srcLayers={"pool1"};
run;

deepLearn.addLayer /
   layer={type="POOL"
          width=2
          height=2
          stride=2
          pool='max'
         }
   modelTable={name="LeNet"}
```

```
   name="pool2"
   srcLayers={"conv2"};
run;

deepLearn.addLayer /
   layer={type="FC"
          n=500
          act='relu'
          init="XAVIER"
         }
   modelTable={name="LeNet"}
   name="ip1"
   srcLayers={"pool2"};
run;

deepLearn.addLayer /
   layer={type="FC"
          n=10
          act='identity'
          init="XAVIER"
         }
   modelTable={name="LeNet"}
   name="ip2"
   srcLayers={"ip1"};
run;

deepLearn.addLayer /
   layer={type="OUTPUT"
          act='softmax'
         }
   modelTable={name="LeNet"}
   name="loss"
   srcLayers={"ip2"};
run;
```

After layers have been added to the model, image data is loaded from the directory that you specify. Next, a preexisting model weights file is loaded from the directory that you specify. Finally, the model table is scored using the image data and model weights that you previously loaded.

```
image.loadImages  /
   casOut={name="myImages",
           replace=TRUE
          }
   path="/u/userDir/directory_with_images";
run;

deepLearn.dlImportModelWeights /
   formatType="CAFFE"
   modelTable={name="LeNet"}
   modelWeights={name="lenetWeights",
                 replace=TRUE
                }
   weightFilePath="/u/userDir/model_weight_directory/model_weight_file";
run;

deepLearn.dlScore /
```

```
      initWeights={name="lenetWeights"}
      modelTable={name="LeNet"}
      table={name="myImages"};
   run;

quit;
```

Example 5: Assign Target Label Information

You can use CASL to assign target label information using the dlLabelTarget action.

The following example shows how you can use CASL to assign target label information using the dlLabelTarget action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                             /* 2 */
   session casauto;

   <Additional steps before assigning                 /* 3 */
    target label information>

   deepLearn.dlLabelTarget /                          /* 4 */
      initWeights={name="IW_table_name"}
      labelTable={name="Label_table_name"}
      modelTable={name="Model_table_name"}
      modelWeights={name="MW_table_name"};
   run;

quit;                                                 /* 5 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, make sure that you have built a model, an in-memory table that contains model weights, and a table that contains the target label information when training a classification model. For more information about training a model, see Train a Deep Learning Model on page 56.

4 Calls the dlLabelTarget action in the Deep Learning action set using the in-memory table that contains the model weights, the table that contains the target label information when training a classification model, the in-memory table that is the model, and the in-memory table that is used to store the model weights.

   Note: You need to modify the `name` values for the `initWeights`, `labelTable`, `modelTable`, and `modelWeights` parameters to the fit the model weights in-memory table, the target label information table, the in-memory model table, and the in-memory table that is used to store the model weights.

5 Ends the code to run a CAS action in SAS.

To illustrate the possible steps that you might need to take before assigning target label information, consider the following example. In it, a library is added, and a table is loaded with sample data. Next, an empty recurrent neural network model is created, and layers are added to it.

```
proc cas;

   session casauto;

   table.addCaslib /
      datasource={srcType="PATH"}
      name="My_library_name"
      path="/u/userDir/directory_with_sample_data";
   run;

   table.loadTable result=r /
      caslib="My_library_name"
      importOptions={fileType="CSV"}
      path="My_file_name"
      casOut={name="CAS_file_name", replace=true};
   run;

   deepLearn.buildModel /
      modelTable={name="mtRnn",
                  replace=TRUE
                 }
      type="RNN";
   run;

   deepLearn.addLayer /
      layer={type="INPUT"}
      modelTable={name="mtRnn"}
      name="data";
   run;

   deepLearn.addLayer /
      layer={type="recurrent"
             n=50
             act='sigmoid'
             init='xavier'
             rnnType='gru'
            }
      modelTable={name="mtRnn"}
      name="rnn1"
      srcLayers={"data"};
   run;
```

After layers have been added to the model, you can perform training using the model and the input data. Next, you can modify the target label information for the trained model. Note that the `initWeights` parameter in the dlLabelTarget action uses the table that was created by the `modelWeights` parameter in the dlTrain action. The `modelWeights` parameter in the dlLabelTarget action specifies the name of your new model weights file. Also, you might need to load the table that contains the target label information that you want to use (This is “Label_table_name” in this example) if it is not already in a directory that your CAS actions have access to.

```
   deepLearn.addLayer /
      layer={type="output"
             act='softmax'
```

```
            init='xavier'
           }
      modelTable={name="mtRnn"}
      name="outlayer"
      srcLayers={"rnn1"};
   run;

   deepLearn.dlTrain /
      inputs={{name="Variable_name"}}
      modelTable={name="mtRnn"}
      modelWeights={name="MW_table_name"}
      table={name="CAS_file_name"};
   run;

   deepLearn.dlLabelTarget /
      initWeights={name="MW_table_name"}
      labelTable={name="Label_table_name"}
      modelTable={name="mtRnn"}
      modelWeights={name="New_MW_table_name"};
   run;

quit;
```

Example 6: Score a Table Using a Deep Learning Model

You can use CASL to score a table with a deep learning model using the dlScore action.

The following example shows how you can use CASL to score a table with a deep learning model using the dlScore action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";      /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                              /* 2 */
   session casauto;

   <Additional steps before scoring a model>           /* 3 */

   deepLearn.dlScore /                                 /* 4 */
      initWeights={name="IW_table_name"}
      modelTable={name="Model_table_name"}
      table={name="Table_name"};
   run;

quit;                                                  /* 5 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, make sure that you have trained your model before you score it. For more information about training a model, see Train a Deep Learning Model on page 56.

4 Calls the dlScore action in the Deep Learning action set using the in-memory table that contains the model weights, the in-memory table that is the model, and the input CAS table that you are using to help test your model. The `casOut` and `copyVars` parameters can be specified to store the scored results.

Note: You need to modify the `name` values for the `initWeights`, `modelTable`, and `table` parameters to the fit the model weights in-memory table, the in-memory model table, and the input CAS table that you want to use to test your model.

5 Ends the code to run a CAS action in SAS.

To illustrate the possible steps that you might need to take before scoring a deep learning model, consider the following example. In it, a library is added, and tables are loaded with sample data. Next, an empty recurrent neural network model is created, and layers are added to it.

```
proc cas;

   session casauto;

   table.addCaslib /
      datasource={srcType="PATH"}
      name="My_library_name"
      path="/u/userDir/directory_with_sample_data";
   run;

   table.loadTable result=r /
      caslib="My_library_name"
      importOptions={fileType="CSV"}
      path="My_train_file_name"
      casOut={name="CAS_file_name_train", replace=true};
   run;

   table.loadTable result=r /
      caslib="My_library_name"
      importOptions={fileType="CSV"}
      path="My_test_file_name"
      casOut={name="CAS_file_name_test", replace=true};
   run;

   deepLearn.buildModel /
      modelTable={name="mtRnn",
                  replace=TRUE
                 }
      type="RNN";
   run;

   deepLearn.addLayer /
      layer={type="INPUT"}
      modelTable={name="mtRnn"}
      name="data";
   run;
```

After layers have been added to the model, you can train your model using input data, and score the deep learning model using test data. The actual steps that you should take before scoring a deep learning model

varies depending on your modeling goals. For an example with sample data that you can run, see Build a Deep Learning Model with Tabular Data on page 68.

```
   deepLearn.addLayer /
      layer={type="recurrent"
             n=50
             act='sigmoid'
             init='xavier'
             rnnType='gru'
            }
      modelTable={name="mtRnn"}
      name="rnn1"
      srcLayers={"data"};
   run;

   deepLearn.addLayer /
      layer={type="output"
             act='softmax'
             init='xavier'
            }
      modelTable={name="mtRnn"}
      name="outlayer"
      srcLayers={"rnn1"};
   run;

   deepLearn.dlTrain /
      inputs={{name="Variable_name"}}
      modelTable={name="mtRnn"}
      modelWeights={name="MW_table_name"}
      table={name="CAS_file_name_train"}
      target="target_variable";
   run;

   deepLearn.dlScore /
      casOut={name="train_scored",
              replace=TRUE
             }
      initWeights={name="MW_table_name"}
      modelTable={name="mtRnn"}
      table={name="CAS_file_name_test"};
   run;

quit;
```

Example 7: Train a Deep Learning Model

You can use CASL to train a deep learning model using the dlTrain action.

The following example shows how you can use CASL to train a deep learning model using the dlTrain action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /* 1 */
```

```
cas casauto;
caslib _all_ assign;

proc cas;                                              /* 2 */
   session casauto;

   <Additional steps before training a model>          /* 3 */

   deepLearn.dlTrain /                                 /* 4 */
      inputs={{name="Variable_name"}}
      modelTable={name="Model_table_name"}
      modelWeights={name="MW_table_name"}
      table={name="Table_name"};
   run;

quit;                                                  /* 5 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, make sure that you have built a model and added the layers to it that you want to use before you train your model. For more information about building a model, see Build a Deep Learning Model on page 45. For more information about adding layers to a model, see Add a Layer to a Deep Learning Model on page 43.

4 Calls the dlTrain action in the Deep Learning action set using the input variables that you specify to use in the analysis, the in-memory table that is the model, the name of the in-memory table that you want to use to store the model weights, and the CAS input table that contains the data that you are using to train your model. You can also use the optimizer option to specify optimization parameters.

   Note: You need to modify the `name` values for the `inputs`, `modelTable`, `modelWeights`, and `table` parameters to fit the input variables, the in-memory model table, the model weights in-memory table, and the CAS input table that you want to use.

5 Ends the code to run a CAS action in SAS.

To illustrate the possible steps that you might need to take before training a deep learning model, consider the following example. In it, a library is added, and a table is loaded with sample data. Next, an empty recurrent neural network model is created, and layers are added to it.

```
proc cas;

   session casauto;

   table.addCaslib /
      datasource={srcType="PATH"}
      name="My_library_name"
      path="/u/userDir/directory_with_sample_data";
   run;

   table.loadTable result=r /
      caslib="My_library_name"
      importOptions={fileType="CSV"}
      path="My_file_name"
      casOut={name="CAS_file_name", replace=true};
   run;
```

```
   deepLearn.buildModel /
      modelTable={name="mtRnn",
                  replace=TRUE
                 }
      type="RNN";
   run;

   deepLearn.addLayer /
      layer={type="INPUT"}
      modelTable={name="mtRnn"}
      name="data";
   run;

   deepLearn.addLayer /
      layer={type="recurrent"
             n=50
             act='sigmoid'
             init='xavier'
             rnnType='gru'
            }
      modelTable={name="mtRnn"}
      name="rnn1"
      srcLayers={"data"};
   run;
```

After layers have been added to the model, you can perform training using the model and the input data. The actual steps that you should take before training a deep learning model varies depending on your modeling goals. For an example with sample data that you can run, see Build a Deep Learning Model with Tabular Data on page 68.

```
   deepLearn.addLayer /
      layer={type="output"
             act='softmax'
             init='xavier'
            }
      modelTable={name="mtRnn"}
      name="outlayer"
      srcLayers={"rnn1"};
   run;

   deepLearn.dlTrain /
      inputs={{name="Variable_name"}}
      modelTable={name="mtRnn"}
      modelWeights={name="MW_table_name"}
      table={name="CAS_file_name"};
   run;

quit;
```

Example 8: Tune a Deep Learning Model

You can use CASL to tune a deep learning model using the dlTune action.

The following example shows how you can use CASL to tune a deep learning model using the dlTune action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /*1*/
cas casauto;
caslib _all_ assign;

proc cas;                                             /*2*/
   session casauto;

   <Additional steps before tuning a model>           /*3*/

   deepLearn.dlTune /                                 /*4*/
      inputs={{name="Variable_name"}}
      modelTable={name="Model_table_name"}
      modelWeights={name="MW_table_name"}
      optimizer={algorithm={method="ADAM"}}
      table={name="Table_name"}
      validTable={name="V_table_name"};
   run;

quit;                                                 /*5*/
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, make sure that you have built a model and added the layers to it that you want to use before you tune your model. For more information about building a model, see Build a Deep Learning Model on page 45. For more information about adding layers to a model, see Add a Layer to a Deep Learning Model on page 43.

4 Calls the dlTune action in the Deep Learning action set using the input variables that you specify to use in the analysis, the in-memory table that is the model, the name of the in-memory table that you want to use to store the model weights, optimization options, the CAS input table that contains the data that you are using to tune your model, and the CAS table that contains the validation data that you want to use. The validation table must have the same columns and data types as the training table.

   Note: You need to modify the `name` values for the `inputs`, `modelTable`, `modelWeights`, `table`, and `validTable` parameters to fit the variables, model table, model weights table, CAS input training table, and the CAS input validation data that you want to use.

5 Ends the code to run a CAS action in SAS.

To illustrate how you could tune a deep learning model, consider the following example. In this example, a library is added, and input data is loaded to it. Next, an empty convolutional neural network is created, and then layers are added to it:

```
proc cas;
   session casauto;

   table.addCaslib /
      datasource={srcType="PATH"}
      name="My_library_name"
      path="/u/userDir/directory_with_sample_data";
   run;
```

```
table.loadTable result=r /
   caslib="My_library_name"
   importOptions={fileType="CSV"}
   path="train_data.csv"
   casOut={name="train_data", replace=true};
run;

table.loadTable result=r /
   caslib="My_library_name"
   importOptions={fileType="CSV"}
   path="test_data.csv"
   casOut={name="test_data", replace=true};
run;

deepLearn.buildModel /
   modelTable={name="ConvNet",
               replace=TRUE
              }
   type="CNN";
run;

deepLearn.addLayer /
   layer={type="INPUT"
          nchannels=3
          width=32
          height=32
          scale=1
          offsets={126.02464141, 123.7085042, 114.85431865}
         }
   modelTable={name="ConvNet"}
   name="data";
run;

deepLearn.addLayer /
   layer={type="CONVO"
          nFilters=32
          width=3
          height=3
          stride=1
          init="MSRA2"
         }
   modelTable={name="ConvNet"}
   name="conv1"
   srcLayers={"data"};
run;
```

After layers have been added to the model, the dlTune action uses optimization options to tune hyperparameters for the deep learning model:

```
deepLearn.addLayer /
   layer={type="POOL"
          width=3
          height=3
          stride=2
          pool="AVERAGE"
```

```
            }
      modelTable={name="ConvNet"}
      name="pool3"
      srcLayers={"conv1"};
   run;

   deepLearn.addLayer /
      layer={type="OUTPUT"
             act='SOFTMAX'
             init="MSRA2"
            }
      modelTable={name="ConvNet"}
      name="outlayer"
      srcLayers={"pool3"};
   run;

   deepLearn.dlTune /
      inputs={{name="inputVar"}}
      modelTable={name="ConvNet"}
      modelWeights={name="MW_table_name",
                    replace=TRUE
                   }
      nominals={{name="nominal_variable_name"}}
      optimizer={algorithm={method="ADAM",
                            beta1={dlBeta1Range},
                            beta2={dlBeta2Range},
                            clipGradMax=100,
                            clipGradMin=-100,
                            learningRate={dlLearningRateRange},
                            learningRatePolicy="STEP",
                            gamma={dlGammaRange},
                            stepSize={dlStepSizeRange}
                           },
                 maxEpochs=2,
                 miniBatchSize={value=1},
                 seed=8675309,
                 tuneIter=10
                }
      seed=54321
      table={name="train_data"}
      target="target_variable_name"
      validTable={name="test_data"};
   run;

quit;
```

Example 9: Display Model Information

You can use CASL to display model information using the modelInfo action.

The following example shows how you can use CASL to display model information using the modelInfo action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";      /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                              /* 2 */
   session casauto;

   <Additional steps before displaying                 /* 3 */
    model information>

   deepLearn.modelInfo /                               /* 4 */
      modelTable={name="Model_table_name"};
   run;

quit;                                                  /* 5 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, make sure that you have first built a model before you display model information. For more information about building a model, see Build a Deep Learning Model on page 45.

4 Calls the modelInfo action in the Deep Learning action set using the in-memory table that stores the model graph definition that you want to use.

Note: You need to modify the `name` value of the `modelTable` parameter to the fit the in-memory table that stores the model graph definition that you want to use.

5 Ends the code to run a CAS action in SAS.

To illustrate the possible steps that you might need to take before displaying model information, consider the following example. In it, a library is added, and a table is loaded with sample data. Next, an empty recurrent neural network model is created, and layers are added to it.

```
proc cas;

   session casauto;

   table.addCaslib /
      datasource={srcType="PATH"}
      name="My_library_name"
      path="/u/userDir/directory_with_sample_data";
   run;

   table.loadTable result=r /
      caslib="My_library_name"
      importOptions={fileType="CSV"}
      path="My_file_name"
      casOut={name="CAS_file_name", replace=true};
   run;

   deepLearn.buildModel /
      modelTable={name="mtRnn",
                  replace=TRUE
                 }
      type="RNN";
```

```
   run;

   deepLearn.addLayer /
      layer={type="INPUT"}
      modelTable={name="mtRnn"}
      name="data";
   run;

   deepLearn.addLayer /
      layer={type="recurrent"
             n=50
             act='sigmoid'
             init='xavier'
             rnnType='gru'
            }
      modelTable={name="mtRnn"}
      name="rnn1"
      srcLayers={"data"};
   run;
```

In this example, the modelInfo action is called to display information for the mtRnn table after layers have been added to it.

```
   deepLearn.addLayer /
      layer={type="output"
             act='softmax'
             init='xavier'
            }
      modelTable={name="mtRnn"}
      name="outlayer"
      srcLayers={"rnn1"};
   run;

   deepLearn.modelInfo /
      modelTable={name="mtRnn"};
   run;

quit;
```

Example 10: Remove a Layer from a Deep Learning Model

You can use CASL to remove a layer from a deep learning model using the removeLayer action.

The following example shows how you can use CASL to remove a layer from a deep learning model using the removeLayer action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                             /* 2 */
```

```
   session casauto;

   <Additional steps before removing a layer>          /*3*/

   deepLearn.removeLayer /                             /*4*/
      modelTable={name="Model_table_name"}
      name="Layer_to_remove";
   run;

quit;                                                  /*5*/
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 After you connect to a CAS session and begin the code to run a CAS action, make sure that you have added a layer to your model before you remove one. For more information about adding a layer to your model, see Add a Layer to a Deep Learning Model on page 43.

4 Calls the removeLayer action in the Deep Learning action set using the in-memory model table that you specified, and the name of the layer that you want to remove from the model. Note that the directed acyclic graph needs to be maintained after one layer is removed.

   Note: You need to modify the `name` value for the `modelTable` parameter to specify the in-memory model table that you want to use. You also need to modify the value of the `name` parameter to specify the name of the layer that you want to remove from your model.

5 Ends the code to run a CAS action in SAS.

To illustrate the possible steps that you might need to take before removing a layer, consider the following example. In it, an empty convolutional neural network (CNN) model is created, and layers are added to it:

```
proc cas;
   session casauto;

   deepLearn.buildModel /
      modelTable={name="LeNet",
                  replace=TRUE
                 }
      type="CNN";
   run;

   deepLearn.addLayer /
      layer={type="INPUT"
             nchannels=1
             width=28
             height=28
             scale=0.00392156862745098039
            }
      modelTable={name="LeNet"}
      name="data";
   run;

   deepLearn.addLayer /
      layer={type="CONVO"
             nFilters=20
             width=5
             height=5
```

```
          stride=1
         }
   modelTable={name="LeNet"}
   name="conv1"
   srcLayers={"data"};
run;

deepLearn.addLayer /
   layer={type="POOL"
          width=2
          height=2
          stride=2
         }
   modelTable={name="LeNet"}
   name="pool1"
   srcLayers={"conv1"}
   replace=TRUE;
run;

deepLearn.addLayer /
   layer={type="CONVO"
          nFilters=50
          width=5
          height=5
          stride=1
         }
   modelTable={name="LeNet"}
   name="conv2"
   srcLayers={"pool1"};
run;
```

After you finish adding layers to your model, you can remove them with the removeLayer action. Note that once you remove a layer from your model, you need to update the fully connected layer and maintain the directed acyclic graph before your model will be ready to use for training. For information about how to add layers back once they have been removed, see Build a LeNet Model on page 92.

```
 deepLearn.addLayer /
   layer={type="POOL"
          width=2
          height=2
          stride=2
          pool='max'
         }
   modelTable={name="LeNet"}
   name="pool2"
   srcLayers={"conv2"};
run;

deepLearn.addLayer /
   layer={type="FC"
          n=500
         }
   modelTable={name="LeNet"}
   name="fc1"
   srcLayers={"pool2"};
run;
```

```
   deepLearn.addLayer /
      layer={type="OUTPUT"
             act='softmax'
            }
      modelTable={name="LeNet"}
      name="outlayer"
      srcLayers={"fc1"};
   run;

   deepLearn.removeLayer /
      modelTable={name="LeNet"}
      name="conv2";
   run;

   deepLearn.removeLayer /
      modelTable={name="LeNet"}
      name="pool2";
   run;

quit;
```

Example 11: Load a Sample Data Source for a Deep Learning Model

You can use CASL to load a sample data source for a deep learning model using the Table action set. The optimization methods that are used in deep learning are sensitive to how the data is distributed. In order to achieve the best performance, randomly shuffling your data using the shuffle action in the Table action set and a smaller blocksize for the training table are highly recommended.

The following example shows how you can use CASL to load a sample data source for a deep learning model using the Table action set.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";        /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                                /* 2 */
   session casauto;

   table.addCaslib /                                     /* 3 */
      datasource={srcType="PATH"}
      name="My_library_name"
      path="/u/userDir/directory_with_sample_data";
   run;

   table.loadTable result=r /                            /* 4 */
      caslib="My_library_name"
      importOptions={fileType="CSV"}
      path="My_file_name"
      casOut={name="CAS_file_name", replace=true};
   run;
```

```
   table.tableInfo /                                    /* 5 */
      name="CAS_file_name";
   run;

   table.tableDetails /                                 /* 6 */
     name="CAS_file_name";
   run;

   table.fetch /                                        /* 7 */
      maxRows=20
      table={name="CAS_file_name"};
   run;

quit;                                                   /* 8 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Adds a CAS library in your CAS session. The addCaslib action creates a library with the library name that you specify for the `name` parameter. It fills the library with the files in the directory that you specify using the `path` parameter.

Note: You need to modify the value of the `name` parameter to specify the library name that you want to create. You also need to modify the value of the `path` parameter to provide a path to the directory that contains the sample data files that you want to use. Finally, make sure that the path that you use is one that you can access from a CAS session.

4 Loads the source data into a CAS table using the loadTable action.

Note: You need to modify the value of the `caslib` parameter to specify the name of the CAS library that you created in the previous step. You also need to modify the value of the `path` parameter to specify the filename that you want to use (for example, 'My_data.csv'). Finally, you need to modify the value of the `name` parameter to provide the CAS table name that you want to use (for example, 'My_data'). The CAS table that you specify is loaded with the data from the file that you specified as the value of the `path` parameter. If the file type that you specified for the `path` parameter is different from 'CSV', you also need to modify the `fileType` value for the `importOptions` parameter to match the file type that you are using.

5 Displays overview information about the CAS table that you loaded using the tableInfo action.

Note: You need to modify the value of the `name` parameter to specify the CAS filename that you previously chose for your loaded data.

6 Displays details information about the CAS table that you loaded using the tableDetails action.

Note: You need to modify the value of the `name` parameter to specify the CAS filename that you previously chose for your loaded data.

7 Displays observations in the CAS table that you loaded using the fetch action. This step helps you check that your data has been loaded properly, and limits output to the first 20 observations.

Note: You need to modify the value of the `name` parameter to specify the CAS filename that you previously chose for your loaded data.

8 Ends the code to run a CAS action in SAS.

Note: For a specific example that loads a sample data source, see Build a Deep Learning Model with Tabular Data on page 68.

Example 12: Build a Deep Learning Model with Tabular Data

You can use CASL to build a deep learning model with tabular data.

This example shows how you can use CASL to build a deep learning model with tabular data using the HMEQ sample data set (hmeq.csv) that contains information about loans.

The HMEQ data set includes home equity information about 5,960 fictitious mortgages. Each observation in this file represents an applicant for a home equity loan, and all applicants have an existing mortgage. In the HMEQ data set, the binary variable `BAD` indicates whether an applicant eventually defaulted or was ever seriously delinquent with a value of 1.

You can download hmeq.csv from the SAS support website. To download this file, go to the SAS Viya Knowledge Base Documentation Page.

After you download the hmeq.csv file, split it into two equal CSV files with 2,980 observations in each. In this example, the file with the first 2,980 observations is called hmeq_train.csv, and the file with the second 2,980 observations is called hmeq_test.csv.

Add the hmeq_train.csv and hmeq_test.csv files to a location that you can access on your server network.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";        /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                                /* 2 */
   session casauto;

  table.addCaslib /                                      /* 3 */
      datasource={srcType="PATH"}
      name="casdata"
      path="/u/userDir/directory_with_sample_data";
   run;

   table.loadTable result=r /                            /* 4 */
      caslib="casdata"
      importOptions={fileType="CSV"}
      path="hmeq_train.csv"
      casOut={name="hmeq_train", replace=true};
   run;

   table.loadTable result=r /                            /* 5 */
      caslib="casdata"
      importOptions={fileType="CSV"}
      path="hmeq_test.csv"
      casOut={name="hmeq_test", replace=true};
   run;
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Adds the `casdata` library with the files at the specified directory path.

Note: You need to modify the value of the `path` parameter to specify the path to the files that you want to use.

4 Loads the hmeq_train CAS table from the hmeq_train.csv file in the `casdata` library.

5 Loads the hmeq_test CAS table from the hmeq_test.csv file in the `casdata` library.

Here are the results from adding the `casdata` library:

The SAS System

Results from table.addCaslib

CAS Library Information

Library	Type	Path	Sub-directories included	Session local	Active
casdata	PATH		No	Yes	Yes

After you load the data that you want to use, you can use actions in the Table action set to review your data, including overview information, details, and observations:

```
table.tableInfo /                                    /* 1 */
   name="hmeq_train";
run;

table.tableDetails /                                 /* 2 */
  name="hmeq_train";
run;

table.fetch /                                        /* 3 */
   maxRows=20
   table={name="hmeq_train"};
run;
```

1 Uses the tableInfo action in the Table action set to display overview table information for the hmeq_train CAS table.

2 Uses the tableDetails action in the Table action set to display details table information for the hmeq_train CAS table.

3 Uses the fetch action in the Table action set to display the first 20 rows of the hmeq_train CAS table.

Here is the overview table information for the hmeq_train CAS table:

The SAS System

Results from table.tableInfo

Table Information for Caslib casdata

Table Name	Number of Rows	Number of Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Source Name	Source Caslib	Compressed
HMEQ_TRAIN	2980	13	utf-8	20Sep2017:18:21:24	20Sep2017:18:21:24	No	No	No	hmeq_train.csv	casdata	No

Here is the details table information for the hmeq_train CAS table:

The SAS System

Results from table.tableDetails

Detail Information for hmeq_train in Caslib casdata.

Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated
ALL	284	142	2980	392369	34769	142	417840	142	417840	0	0

Here is the first 20 rows of the hmeq_train CAS table:

The SAS System

Results from table.fetch

Selected Rows from Table HMEQ_TRAIN

Index	BAD	LOAN	MORTDUE	VALUE	REASON	JOB	YOJ	DEROG	DELINQ	CLAGE	NINQ	CLNO	DEBTINC
1	1	1100	25860	39025	HomeImp	Other	10.5	0	0	94.36666667	1	9	.
2	1	1300	70053	68400	HomeImp	Other	7	0	2	121.8333333	0	14	.
3	1	1500	13500	16700	HomeImp	Other	4	0	0	149.4666667	1	10	.
4	1	1500	.	.			.	.	.	.	.	.	.
5	0	1700	97800	112000	HomeImp	Office	3	0	0	93.33333333	0	14	.
6	1	1700	30548	40320	HomeImp	Other	9	0	0	101.4660019	1	8	37.11361356
7	1	1800	48649	57037	HomeImp	Other	5	3	2	77.1	1	17	.
8	1	1800	28502	43034	HomeImp	Other	11	0	0	88.76602988	0	8	36.88489409
9	1	2000	32700	46740	HomeImp	Other	3	0	2	216.9333333	1	12	.
10	1	2000	.	62250	HomeImp	Sales	16	0	0	115.8	0	13	.
11	1	2000	22608	.			18	.	.	.	.	.	.
12	1	2000	20627	29800	HomeImp	Office	11	0	1	122.5333333	1	9	.
13	1	2000	45000	55000	HomeImp	Other	3	0	0	86.06666667	2	25	.
14	0	2000	64536	87400		Mgr	2.5	0	0	147.1333333	0	24	.
15	1	2100	71000	83850	HomeImp	Other	8	0	1	123	0	16	.
16	1	2200	24280	34687	HomeImp	Other	.	0	1	300.8666667	0	8	.
17	1	2200	90957	102600	HomeImp	Mgr	7	2	6	122.9	1	22	.
18	1	2200	23030	.			19	.	.	.	.	.	3.711312299
19	1	2300	28192	40150	HomeImp	Other	4.5	0	0	54.6	1	16	.
20	0	2300	102370	120953	HomeImp	Office	2	0	0	90.99253347	0	13	31.58850318

Similarly, you can check that the data that you plan to use to test your model has been properly loaded using actions in the Table action set:

```
table.tableInfo /                                    /* 1 */
  name="hmeq_test";
run;

table.tableDetails /                                 /* 2 */
  name="hmeq_test";
run;

table.fetch /                                        /* 3 */
   maxRows=20
   table={name="hmeq_test"};
run;
```

1. Uses the tableInfo action in the Table action set to display overview table information for the hmeq_test CAS table.
2. Uses the tableDetails action in the Table action set to display details table information for the hmeq_test CAS table.
3. Uses the fetch action in the Table action set to display the first 20 rows of the hmeq_test CAS table.

Here is the overview table information for the hmeq_test CAS table:

The SAS System

Results from table.tableInfo

Table Information for Caslib casdata

Table Name	Number of Rows	Number of Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Source Name	Source Caslib	Compressed
HMEQ_TEST	2980	13	utf-8	20Sep2017:18:21:25	20Sep2017:18:21:25	No	No	No	hmeq_test.csv	casdata	No

Here is the details table information for the hmeq_test CAS table:

The SAS System

Results from table.tableDetails

Detail Information for hmeq_test in Caslib casdata.

Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated
ALL	284	142	2980	392965	35365	142	418416	142	418416	0	0

Here is the first 20 rows of the hmeq_test CAS table:

The SAS System

Results from table.fetch

Selected Rows from Table HMEQ_TEST

Index	BAD	LOAN	MORTDUE	VALUE	REASON	JOB	YOJ	DEROG	DELINQ	CLAGE	NINQ	CLNO	DEBTINC
1	0	16300	66038	98585	DebtCon	Office	1	0	0	162.0654119	3	19	32.75346067
2	0	16400	50941	78801	DebtCon	ProfExe	4	0	0	266.040799	2	32	32.90435701
3	0	16400	74904	92308	DebtCon	Other	6	0	0	202.4904448	3	19	42.42034531
4	0	16400	62786	79154	DebtCon		7	.	.	.	.	.	24.83894007
5	0	16400	21857	51994	HomeImp	ProfExe	7	0	0	201.9333333	0	14	.
6	0	16400	71264	94726	DebtCon	Other	1	0	0	177.2573424	1	31	35.03448136
7	0	16400	119879	128546	DebtCon		16	0	0	182.9973812	0	17	40.85088684
8	0	16400	86623	124048			.	.	.	.	.	.	35.81076936
9	1	16400	54818	74447	DebtCon	Office	3	2	3	208.0888643	1	50	41.95642972
10	0	16400	57139	164133	DebtCon	ProfExe	21	0	0	238.4610031	0	27	22.20684601
11	0	16400	73466	89453	DebtCon	Other	4	0	0	198.8274238	3	19	41.52349741
12	0	16400	63574	88586	HomeImp	Other	.	0	0	298.1465517	0	20	29.15337202
13	0	16400	82869	94390	DebtCon	ProfExe	8	0	2	358.0506602	.	42	39.06062731
14	0	16400	57651	76854	DebtCon	Office	1	0	0	172.2000922	0	25	31.81962217
15	1	16400	48700	72500	DebtCon	Other	.	0	1	127.3	1	39	.
16	0	16400	81700	109025	DebtCon	ProfExe	10	0	0	155.2666667	3	19	.
17	0	16400	62018	91832	DebtCon	ProfExe	9	0	0	202.3225002	0	19	20.38046421
18	0	16400	.	51454	DebtCon	Other	.	.	.	338.1108068	.	17	33.49183609
19	0	16400	107669	115881	DebtCon	Other	0	0	0	136.8121195	1	21	39.69996168
20	0	16400	47428	76483	DebtCon	ProfExe	4	0	0	232.3155447	3	31	33.10683578

After you check that your data has been loaded properly, you can build an empty recurrent neural network model and view information about it using actions in the Deep Learning action set:

```
deepLearn.buildModel /                          /*1*/
   modelTable={name="hmeqRnn",
               replace=TRUE
              }
   type="RNN";
run;

deepLearn.modelInfo /                           /*2*/
   modelTable={name="hmeqRnn"};
run;
```

1 Uses the buildModel action in the Deep Learning action set to create an empty model for building a recurrent neural network. In this example, the empty model is given the name hmeqRnn.

2 Uses the modelInfo action in the Deep Learning action set to display model information about the hmeqRnn model that was created in the previous step.

Here is the results from building the empty hmeqRnn model:

The SAS System

Results from deepLearn.buildModel

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	hmeqrnn	1	5

Here is the model information for the hmeqRnn model:

The SAS System

Results from deepLearn.modelInfo

Model hmeqrnn Information Details	
Model Name	hmeqrnn
Model Type	Recurrent Neural Network
Number of Layers	0
Number of Input Layers	0
Number of Output Layers	0
Number of Convolutional Layers	0
Number of Pooling Layers	0
Number of Fully Connected Layers	0

Once you have built an empty recurrent neural network model, you can add layers to it using actions in the Deep Learning action set:

```
deepLearn.addLayer /                            /*1*/
   layer={type="INPUT"}
   modelTable={name="hmeqRnn"}
   name="data";
run;
```

```
deepLearn.addLayer /                          /* 2 */
   layer={type="recurrent"
          n=50
          act='sigmoid'
          init='xavier'
          rnnType='gru'
         }
   modelTable={name="hmeqRnn"}
   name="rnn1"
   srcLayers={"data"};
run;

deepLearn.addLayer /                          /* 3 */
   layer={type="output"
          act='softmax'
          init='xavier'
         }
   modelTable={name="hmeqRnn"}
   name="outlayer"
   srcLayers={"rnn1"};
run;
```

1 Uses the addLayer action in the Deep Learning action set to add an input layer to the hmeqRnn model.

2 Uses the addLayer action in the Deep Learning action set to add a recurrent layer to the hmeqRnn model.

3 Uses the addLayer action in the Deep Learning action set to add an output layer to the hmeqRnn model.

Here is the results from adding the input layer to the hmeqRnn model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	hmeqrnn	10	5

Here is the results from adding the recurrent layer to the hmeqRnn model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	hmeqrnn	23	5

Here is the results from adding the output layer to the hmeqRnn model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	hmeqrnn	35	5

Once you have added layers to your model, you can view information about it to see how it has changed:

```
deepLearn.modelInfo /                          /*1*/
   modelTable={name="hmeqRnn"};
run;

table.fetch /                                  /*2*/
   table={name="hmeqRnn"};
run;
```

1 Uses the modelInfo action in the Deep Learning action set to display overview information about the hmeqRnn model.

2 Uses the fetch action in the Table action set to display details information about the hmeqRnn model.

Here is the overview information about the hmeqRnn model:

The SAS System

Results from deepLearn.modelInfo

Model hmeqrnn Information Details	
Model Name	hmeqrnn
Model Type	Recurrent Neural Network
Number of Layers	3
Number of Input Layers	1
Number of Output Layers	1
Number of Convolutional Layers	0
Number of Pooling Layers	0
Number of Fully Connected Layers	0
Number of Recurrent Layers	1

Here is additional details information about the hmeqRnn model:

The SAS System

Results from table.fetch

Selected Rows from Table HMEQRNN

Index	_DLKey0_	_DLKey1_	_DLChrVal_	_DLNumVal_	_DLLayerID_
1	data	inputopts.dropout	dropout	0	0
2	outlayer	srclayers.0	rnn1	1	2
3	rnn1	rnnopts.reversed	reversed	0	1
4	outlayer	outputopts.init	Xavier	1	2
5	outlayer	outputopts.no_bias	no_bias	0	2
6	hmeqrnn	modeltype	Recurrent Neural Network	3	.
7	data	inputopts.height	height	0	0
8	rnn1	rnnopts.act	Logistic	2	1
9	outlayer	outputopts.std	std	1	2
10	outlayer	outputopts.mean	mean	0	2
11	data	inputopts.nchannels	nchannels	1	0
12	data	inputopts.width	width	0	0
13	outlayer	outputopts.truncfact	truncfact	0	2
14	data	inputopts.flip	No flipping	1	0
15	outlayer	outputopts.act	Softmax	7	2
16	data	inputopts.std	No Standardization	2	0
17	rnn1	rnnopts.n	n	50	1
18	data	layertype	Input Layer	1	0
19	rnn1	rnnopts.rnntype	Gated Recurrent Unit	2	1
20	data	inputopts.scale	scale	1	0

Once you have reviewed your model, you can train and score it using the data sets that you have previously loaded:

```
deepLearn.dlTrain /                              /* 1 */
   inputs={"mortdue", "value", "debtinc"}
   modelTable={name="hmeqRnn"}
   modelWeights={name="hmeqTrainedWeights",
                 replace=TRUE
                }
   table={name="hmeq_train"}
   target="bad";
run;

deepLearn.dlScore /                              /* 2 */
   casOut={name="train_scored",
           replace=TRUE
          }
   initWeights={name="hmeqTrainedWeights"}
   modelTable={name="hmeqRnn"}
   table={name="hmeq_test"};
run;
```

1 Uses the dlTrain action in the Deep Learning action set to train the hmeqRnn model using the hmeq_train CAS data set, and the `mortdue`, `value`, and `debtinc` variables.

2 Uses the dlScore action in the Deep Learning action set to score the trained model from the previous step using the hmeq_test CAS data set.

Here is the results from training the hmeqRnn model using the hmeq_train CAS data set:

The SAS System

Results from deepLearn.dlTrain

Model hmeqrnn Information Details	
Model Name	hmeqrnn
Model Type	Recurrent Neural Network
Number of Layers	3
Number of Input Layers	1
Number of Output Layers	1
Number of Convolutional Layers	0
Number of Pooling Layers	0
Number of Fully Connected Layers	0
Number of Recurrent Layers	1
Number of Weight Parameters	7700
Number of Bias Parameters	151
Total Number of Model Parameters	7851
Approximate Memory Cost for Training (MB)	1

Optimization History of Deep Learning Model for HMEQ_TRAIN			
Epoch	Learning Rate	Loss	Fit Error
0	0.001	0.3811666667	0.762333

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	hmeqTrainedWeights	7819	3

Here is the results from scoring the hmeqRnn model using the hmeq_test CAS data set:

The SAS System

Results from deepLearn.dlScore

Score Information for HMEQ_TEST	
Number of Observations Read	2283
Number of Observations Used	2283
Mean Squared Error	0.926413
Loss Error	0.463206

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	train_scored	2283	1

After you have trained and scored you model, you can experiment with different options, values, or variables to try to improve your model. The following example uses the `delinq` variable, which tracks whether a customer has been delinquent in a mortgage payment before:

```
deepLearn.dlTrain /                                    /* 1 */
   inputs={"mortdue", "value", "delinq"}
   modelTable={name="hmeqRnn"}
   modelWeights={name="hmeqTrainedWeights",
                 replace=TRUE
```

```
                }
   table={name="hmeq_train"}
   target="bad";
run;

deepLearn.dlScore /                                    /*2*/
   casOut={name="train_scored",
           replace=TRUE
          }
   initWeights={name="hmeqTrainedWeights"}
   modelTable={name="hmeqRnn"}
   table={name="hmeq_test"};
run;
```

1 Uses the dlTrain action in the Deep Learning action set to train the hmeqRnn model using the hmeq_train CAS data set, and the `mortdue`, `value`, and `delinq` variables.

2 Uses the dlScore action in the Deep Learning action set to score the trained model from the previous step using the hmeq_test CAS data set.

Here is the results from training the hmeqRnn model using the hmeq_train CAS data set with different input variables:

The SAS System

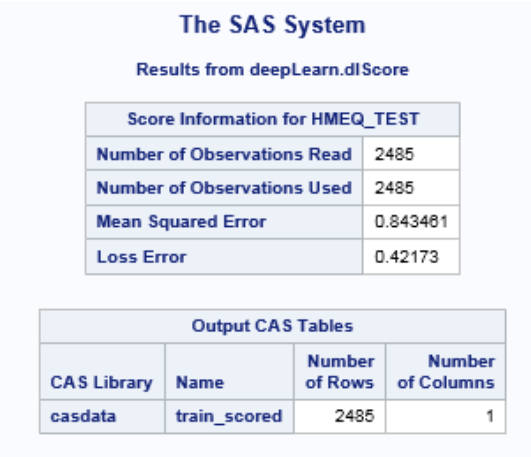
Results from deepLearn.dlTrain

Model hmeqrnn Information Details	
Model Name	hmeqrnn
Model Type	Recurrent Neural Network
Number of Layers	3
Number of Input Layers	1
Number of Output Layers	1
Number of Convolutional Layers	0
Number of Pooling Layers	0
Number of Fully Connected Layers	0
Number of Recurrent Layers	1
Number of Weight Parameters	7700
Number of Bias Parameters	151
Total Number of Model Parameters	7851
Approximate Memory Cost for Training (MB)	1

Optimization History of Deep Learning Model for HMEQ_TRAIN			
Epoch	Learning Rate	Loss	Fit Error
0	0.001	0.3763109192	0.752622

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	hmeqTrainedWeights	7819	3

Here is the results from scoring the hmeqRnn model using the hmeq_test CAS data set with different input variables:



The SAS System

Results from deepLearn.dlScore

Score Information for HMEQ_TEST	
Number of Observations Read	2485
Number of Observations Used	2485
Mean Squared Error	0.843461
Loss Error	0.42173

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	train_scored	2485	1

When training your model, you can use the `optimizer` parameter to specify the settings for the optimization algorithm, optimization mode, and other settings such as a seed, the maximum number of epochs, and so on. For more information about the `optimizer` parameter, see Train a Model on page 105 and Tune a Model on page 107. After you have configured your model the way you want it, you can export your model and view information about it:

```
   deepLearn.dlExportModel /                          /* 1 */
      casout={name="My_Model"}
      initWeights={name="hmeqTrainedWeights"}
      modelTable={name="hmeqRnn"};
   run;

   table.tableInfo /                                  /* 2 */
      name="My_Model";
   run;

   table.tableDetails /                               /* 3 */
     name="My_Model";
   run;

quit;                                                 /* 4 */
```

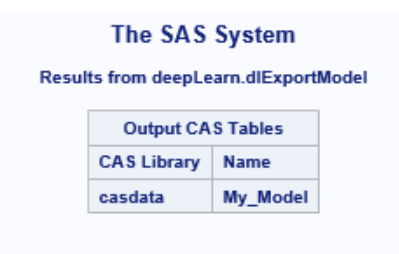
1 Uses the dlExportModel action in the Deep Learning action set to export your finished model. In this example, the model is called My_Model.

2 Uses the tableInfo action in the Table action set to view information about the exported model in the previous step.

3 Uses the tableDetails action in the Table action set to view details information about your exported model.

4 Ends the code to run a CAS action in SAS.

Here is the results from exporting the model using the dlExportModel action:



The SAS System

Results from deepLearn.dlExportModel

Output CAS Tables	
CAS Library	Name
casdata	My_Model

Here is the overview table information about the exported model:

The SAS System

Results from table.tableInfo

Table Information for Caslib casdata

Table Name	Number of Rows	Number of Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Compressed
MY_MODEL	1	2	utf-8	20Sep2017:18:21:43	20Sep2017:18:21:43	No	Yes	No	No

Here is the details table information about the exported model:

The SAS System

Results from table.tableDetails

Detail Information for My_Model in Caslib casdata.

Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated
ALL	142	142	142	5393160	5389752	142	5418720	0	0	0	0

Example 13: Build a Recurrent Neural Network Model

You can use CASL to build a recurrent neural network (RNN) model.

This example shows how you can use CASL to build an RNN model using an input data set that contains text excerpts from approximately 600 brief news articles, and an input data set that contains vector representations for words.

In this example, the news data set has been split into two different files for training and testing. The training data set is called news_train_num.csv, and the test data set is called news_test_num.csv. The input data set that contains vector representations for words is called glove_med_100d_xmax50.txt.

Word embeddings are numeric vector representations of words that are trained based on a large corpus, typically without any label information. Popular word embeddings include Word2Vec, Glove, and FastText. These are publicly available for download. When using word embeddings, any column name can be used in the embedding table, but the first column of the embedding table must be the word and the rest of the columns are treated as numeric vector representations of that word.

The Deep Learning action set does not provide task-oriented word embedding training. Since task-oriented word embedding training requires a very large amount of labeled data to achieve better results, we recommend that you use pre-trained word embeddings like Word2Vec, Glove, or FastText.

If you want to experiment with vector representations for words, one source that you could try using is https://nlp.stanford.edu/projects/glove/.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";        /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                                /* 2 */
```

```
session casauto;

table.addCaslib /                                    /* 3 */
   datasource={srcType="PATH"}
   name="casdata"
   path="/u/userDir/directory_with_sample_data";
run;

table.loadTable result=r /                           /* 4 */
   caslib="casdata"
   importOptions={fileType="CSV"}
   path="news_train_num.csv"
   casOut={name="news_train", replace=true};
run;

table.loadTable result=r /                           /* 5 */
   caslib="casdata"
   importOptions={fileType="CSV"}
   path="news_test_num.csv"
   casOut={name="news_test", replace=true};
run;

table.loadTable result=r /                           /* 6 */
   caslib="casdata"
   importOptions={fileType="delimited",
                  delimiter=' ',
                  getNames=true,
                  guessRows=2.0,
                  varChars=true
                 }
   path="glove_med_100d_xmax50.txt"
   casOut={name="glove", replace=true};
run;
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Adds the `casdata` library with the files at the specified directory path.

Note: You need to modify the value of the `path` parameter to specify the path to the files that you want to use.

4 Loads the news_train CAS table from the news_train_num.csv file in the `casdata` library.

Note: The news_train_num.csv file contains 537 observations. Each observation contains an excerpt of text from a news article and either falls into the category of computer graphics, hockey, or medical issues. In the data set, the nominal variable `text` contains the text of the news article. The binary variable `graphics` indicates whether the document belongs to the computer graphics category (1-yes, 0-no). The binary variable `hockey` indicates whether the document belongs to the hockey category (1-yes, 0-no). The binary variable `medical` indicates whether the document belongs to the medical issues category (1-yes, 0-no). And the nominal variable `newsgroup` contains the group that a news article fits into.

5 Loads the news_test CAS table from the news_test_num.csv file in the `casdata` library.

Note: The news_test_num.csv file contains 60 observations. It has the same variables as the news_train_num.csv file, and is used to test the trained model.

6 Loads the glove CAS table from the glove_med_100d_xmax50.txt file in the `casdata` library.

Note: The glove_med_100d_xmax50.txt file contains vector representations for words and is used during training and scoring.

Here are the results from adding the `casdata` library:

The SAS System

Results from table.addCaslib

CAS Library Information

Library	Type	Path	Sub-directories included	Session local	Active
casdata	PATH		No	Yes	Yes

After you load the data that you want to use, you can use actions in the Table action set to review your data, including overview information, details, and observations:

```
table.tableInfo /                                    /* 1 */
   name="news_train";
run;

table.tableDetails /                                 /* 2 */
  name="news_train";
run;

table.fetch /                                        /* 3 */
   maxRows=10
   table={name="news_train"};
run;
```

1 Uses the tableInfo action in the Table action set to display overview table information for the news_train CAS table.

2 Uses the tableDetails action in the Table action set to display details table information for the news_train CAS table.

3 Uses the fetch action in the Table action set to display the first 10 rows of the news_train CAS table.

Here is the overview table information for the news_train CAS table:

The SAS System

Results from table.tableInfo

Table Information for Caslib casdata

Table Name	Number of Rows	Number of Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Source Name	Source Caslib	Compressed
NEWS_TRAIN	537	7	utf-8	21Oct2017:01:31:05	21Oct2017:01:31:05	No	No	No	news_train_num.csv	casdata	No

Here is the details table information for the news_train CAS table:

The SAS System

Results from table.tableDetails

Detail Information for news_train in Caslib casdata.											
Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated
ALL	282	141	537	681254	642590	141	708536	141	708536	0	0

Here is the first 10 rows of the news_train CAS table:

The SAS System

Results from table.fetch

Selected Rows from Table NEWS_TRAIN

Index	text	graphics	hockey	medical	newsgroup	key	numtarget
1	rocko@wrigley.corp.sgi.com (Richard Casares) writes: Well, suffice to say that it is a sport for those able to make the substantial investment in equipment, etc. But here's something, do you think that the availability of in-line skates and road hockey could contribute to a rise in awareness of ice hockey? I would argue this is having an effect here. Kids play ice hockey in the winter and road hockey in the summer with in-line skates.	0	1	0	hockey	370	1
2	In article < 1ppae1$bt0@bigboote.WPI.EDU> susan@wpi.WPI.EDU (susan) writes: I'll probably get flamed for this, but when I was a kid we would go to my uncles cabin on Middle Bass Island on Lake Erie. We always came home with a nasty case of jiggers (large red bumps where the buggers had burrowed into the skin). My mother would paint the bumps with clear finger nail polish. This was repeated daily for about a week or so. The application of the polish is supposed to suffocate them as it seals of the skin. All I can say is it worked for us. One word of caution though. Putting finger nail polish on a jigger bite stings like hell. (If I do get flamed for this just put jam in my pockets and call me toast.) -- ________ kentcomm!jperkaki@aldhfn.akron.oh.us (and) kentcomm!jperkaki@legend.akron.oh.us	0	0	1	medical	434	0
3	In article < 1993Apr5.212016.25066@eng.cam.ac.uk> dscy@eng.cam.ac.uk (D.S.C. Yap) writes: They were, and even if Washington might consider Patty a bust, I'd rework that trade in a minute. Druce has been a complete and utter bust here, only 5 goals. Daryl Turner : r.s.h contact for the Winnipeg Jets Internet: umturne4@ccu.umanitoba.ca FidoNET: 1:348/701 -or- 1:348/4 (please route through 348/700) Tkachuk over to Zhamnov, up to Sel{nne, he shoots, he scores! The Jets win the Cup! The Jets win the Cup! Essensa for Vezina! Housley for Norris! Sel{nne for Calder!	0	1	0	hockey	234	1
4	Article-I.D.: cbnewsh.1993Apr6.141123.8101 References: < 16BA3A61C.BAHAYES@bcsc02.gov.bc.ca> < 1993Apr2.175062.22810@thunder.mcrcim.mcgill.edu> Distribution: na Organization: AT&T Lines: 21 In article < 1993Apr2.175062.22810@thunder.mcrcim.mcgill.edu> , breton@McRCIM.McGill.EDU (Pierre Breton) writes: Pierre, For purposes of the tie breaker, you only count the first three games in each city. Therefore, Quebec cannot possibly be ahead of Montreal 4-3, and there's probably only one game that counts remaining between Boston and Quebec, which means Boston has probably already won. pete clark - rsh FLYERS contact	0	1	0	hockey	236	1
5	In article < 1993Apr15.161112.21772@cs.rochester.edu> , fulk@cs.rochester.edu (Mark Fulk) writes: |> I don't think " extra-scientific" is a very useful phrase in a discussion |> of the boundaries of science, except as a proposed definiens. Extra-rational |> is a better phrase. In fact, there are quite a number of well-known cases |> of extra-rational considerations driving science in a useful direction. Yeah, but the problem with holding up the " extra-rational" examples as exemplars, or as refutations of well founded methodology, is that you run smack up against such unuseful directions as Lysenko. Such " extra- rational" cases are curiosities -- not guides to methodology. -- Gary H. Merrill [Principal Systems Developer, C Compiler Development] SAS Institute Inc. / SAS Campus Dr. / Cary, NC 27513 / (919) 677-8000 sasghm@theseus.unx.sas.com ... !mcnc!sas!sasghm	0	0	1	medical	542	0
6	In article < 93106.082502ACPS6992@RyeVm.Ryerson.Ca> Raj Ramnarace < ACPS6992@RyeVm.Ryerson.Ca> writes: I thought the red light went on...thus, in the review, the presumption would be to find conclusive evidence that the puck did not go in the net...from the replays I say, even from the rear, the evidence wasn't conclusive that the puck was in or out...in my opinion... Gerald	0	1	0	hockey	351	1
7	At the Lester Patrick Awards lunch, Bill Torrey mentioned that one of his options next season is to be president of the Miami team, with Bob Clarke working for him. At the same dinner, Clarke said that his worst mistake in Philadelphia was letting Mike Keenan go -- in retrospect, almost all players came realize that Keenan knew what it took to win. Rumours are now circulating that Keenan will be back with the Flyers. Nick Polano is sick of being a scapegoat for the schedule made for the Red Wings. After all, Bryan Murray approved it. Gerry Meehan and John Muckler are worried over the Sabres' prospects; Assistant Don Lever says that the Sabres have to get their share now, because a Quebec dynasty is emerging ... The Mighty Ducks have declared that they will not throw money around loosely to buy a team. Oilers coach Ted Green remarked that " There some guys around who can fill Tie Domi's skates, but none who can fill his helmet." Senators' Andrew McBain told off a security guard at Chicago Stadium who warned him of the stairs leading down to the locker room; McBain mouthed off at him, after all being a seasoned professional ... and tumbled down the entire steep flight. gld -- ~~~~~~~~~~~~~~~~~~~~~~~~~~ Je me souviens ~~~~~~~~~~~~~~~~~~~~~~~~~~ Gary L. Dare	0	1	0	hockey	257	1
8	In article < 1993Apr5.203552.1@keen.ucs.mun.ca> slegge@keen.ucs.mun.ca writes: Obviously some reporter for the Ottawa Sun got taken by an April Fools joke...probably started by someone with the Nordiques or the Bruins. Like for example...who is going to reimburse the Flyers for the $15 million they paid to the Nordiques...like the Senators are going to get Lindros and $15 million. The Flyers sent the equivalent of 6 or 7 players (when you include the draft choices) to Quebec, and they are going to get only four back. Some reporter was had real badly and someone must be having a real good laugh seeing as how the so much of the sports media has chosen to publicize this utter nonsense. Can you think...it cannot possibly be true...no need for the " #" ! I cannot believe that anyone would consider giving such crap even the remotest consideration. Gerald	0	1	0	hockey	244	1
9	lor@cbnewsk.cb.att.com (edward.lor) writes: That is what my point really was. There is not straight dependency between the +/- and scored points. Apparently most of the netters have realized it by themselves.	0	1	0	hockey	346	1
10	In article < 1993Apr14.174824.12295@westminster.ac.uk> , kxaes@sun.pol.ac.uk (David Watters) wrote: The best group to keep you informed is the Crohn's and Colitis Foundation of America. I do not know if the UK has a similar organization. The address of the CCFA is CCFA 444 Park Avenue South 11th Floor New York, NY 10016-7374 USA They have a lot of information available and have a number of newsletters. Good Luck. Steve	0	0	1	medical	562	0

Similarly, you can check that the data that you plan to use to test your model has been properly loaded using actions in the Table action set:

```
table.tableInfo /                                        /* 1 */
  name="news_test";
run;

table.tableDetails /                                     /* 2 */
  name="news_test";
run;

table.fetch /                                            /* 3 */
   maxRows=10
   table={name="news_test"};
run;

table.fetch /                                            /* 4 */
   maxRows=20
   table={name="glove"};
run;
```

1 Uses the tableInfo action in the Table action set to display overview table information for the news_test CAS table.

2 Uses the tableDetails action in the Table action set to display details table information for the news_test CAS table.

3 Uses the fetch action in the Table action set to display the first 10 rows of the news_test CAS table.

4 Uses the fetch action in the Table action set to display the first 20 rows of the glove CAS table.

Here is the overview table information for the news_test CAS table:

The SAS System

Results from table.tableInfo

Table Information for Caslib casdata											
Table Name	Number of Rows	Number of Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Source Name	Source Caslib	Compressed
NEWS_TEST	60	7	utf-8	21Oct2017:01:31:23	21Oct2017:01:31:23	No	No	No	news_test_num.csv	casdata	No

Here is the details table information for the news_test CAS table:

The SAS System

Results from table.tableDetails

Detail Information for news_test in Caslib casdata.											
Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated
ALL	120	60	60	62562	58242	60	73320	60	73320	0	0

Here is the first 10 rows of the news_test CAS table:

The SAS System

Results from table.fetch

Selected Rows from Table NEWS_TEST							
Index	text	graphics	hockey	medical	newsgroup	key	numtarget
1	Article-I.D.: news.1993Apr6.204743.21314 References: < rauser.734062606@sfu.ca> Sender: usenet@news.columbia.edu (The Network News) Reply-To: gld@cunixb.cc.columbia.edu (Gary L Dare) Organization: PhDs In The Hall Lines: 17 Nntp-Posting-Host: cunixb.cc.columbia.edu You're right ... I'm sick of seeing all those white guys on skates myself ... the Vancouver Canucks should be half women, and overall one-third Oriental. (-; (-; (-; (-; (-; (-; And I'll gladly volunteer myself for the overage draft. (-; gld -- ------------------------- Je me souviens ------------------------------- Gary L. Dare	0	1	0	hockey	320	1
2	Article-I.D.: oz.1993Apr6.023843.9689 Reply-To: r_turgeo@oz.plymouth.edu (Randy S. Turgeon) Organization: Plymouth State College - Plymouth, N.H. Lines: 13 Here we go again. Is this the same idiot who posted the Gretzky trade to Toronto????: Sheeeesh! You should have waited until we got over that one before this garbage, maybe we would have believed it for half a second (NOT!). By the way, I just heard from Mother Goose that Mario Lemieux was traded to Winnpeg for Tie Domi!!!! Randy PSC (I know noone has heard of it, it's a tiny college in Hickville New Hampshire. Plymouth State College)	0	1	0	hockey	219	1
3	Derian Hatcher's game-misconduct penalty was rescinded by the NHL, allowing the Minnesota defenseman to play in the North Stars' last two regular-season games. Hatcher was given the penalty during a fight at the end of a loss at St. Louis on Sunday, April 11. But the league didn't rescind the game-misconduct penalty Shane Churla received. The Stars recalled center Cal McGowan from their top minor league club in Kalamazoo, Mich., to replace Churla. The above is courtesy of The Washington Times on-line service. Now, here's where I need help. If anyone out there has a tape of Tuesday's Chicago-Minnesota game, please contact me. Terms will be favorable. Also, if anyone can tape tonight's Minnesota-Detroit game, please contact me. This could be quite important. Once again, I will make it worth your trouble. Thanks to all. -- rec.sport.hockey contact for Minnesota North Stars and maintainer of the r.s.h FAQ file Mitch McGowan a.k.a. | KALAMAZOO WINGS | MINNESOTA NORTH STARS | hamlet@u.washington.edu | ST. KILDA SAINTS | TORONTO BLUE JAYS |	0	1	0	hockey	360	1
4	We have been using Iterated Systems compression board to compress pathology images and are getting ratios of 40:1 to 70:1 without too much loss in quality. It is taking about 4 mins per image to compress, on a 25Mhz 486 but decompression is almost real time on a 386 in software alone.	1	0	0	graphics	183	2
5	Article-I.D.: blue.7984 References: < 1993Apr06.133536.20360@watson.ibm.com> Sender: news+@pitt.edu Lines: 20 X-Newsreader: TIN [version 1.1 PL8] jes@watson.ibm.com wrote: : A very simple question : it seems to me that the contraceptive : pill just prevents the ovule to nest in the vagina and forces it to : fall every month. But it does not prevent the fertilization of the : ovule. Is it true ? If yes, is there a risk of extra-uterine : pregnancy, that is the development of the ovule inside the Fallopian : tube ? Actually that is not how the pill works, but it *is* how the IUD works. The oral contraceptive pill actually "prevents" ovulation from occuring by providing negative feedback to the pituitary gland, and thus preventing the LH surge that normally occurs at the time of ovulation. With the IUD what happens is that fertilization may occur, but the device prevents implantation within the wall of the uterus (*not* the vagina). -- =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=|=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-= Kenneth Gilbert __|__ University of Pittsburgh = = General Internal Medicine | "...dammit, not a programmer! = =-=-=-=-=-=-=-=-=-=-=-=-=|=-=-=-=-=-=-=-=-=-=-=-=-=-=	0	0	1	medical	443	0
6	In article < 1993Apr14.102007.20664@uk03.bull.co.uk> , scrowe@hemel.bull.co.uk (Simon Crowe) writes: |> I am looking for an algorithm to determine if a given point is bound by a |> polygon. Does anyone have any such code or a reference to book containing |> information on the subject ? |> |> Regards |> |> Simon Basically, there are two algorithms determining whether a point is inside, outside or on the polygon. The first one is Ray (or half line) method. In this method, you can draw any ray, if the number of the intersection point of the ray and the polygon is even, then it is outside. If the number is odd, then it is inside. Of cause, you have to deal with the special cases which may make you headache. The second method is PI algorithm. Draw the lines between the point and all the vertices on the polygon. Calculate and sum the angles of the successive lines. If the result is 2*PI, then it is inside. If PI, then it is on the polygon. Otherwise it is outside. My experience tells the second method is relible. Hope this helps. Yeh USC	1	0	0	graphics	89	2
7	Has ANYONE who has ordered the new PC version of Imagine ACTUALLY recieved it yet? I'm just about ready to order but reading posts about people still awaiting delivery are making me a little paranoid. Has anyone actually held this piece of software in their own hands? Later, Jim Nobles	1	0	0	graphics	156	2
8	Article-I.D.: adobe.1993Apr15.224049.15616 References: < 1993Apr15.163133.25634@ntmtv> Sender: usenet@adobe.com (USENET NEWS) Organization: Adobe Systems Incorporated Lines: 12 In article < 1993Apr15.163133.25634@ntmtv> janet@ntmtv.com (Janet Jakstys) writes: I've gotten migraines after exercise, though for me it seems to be related to exercising without having eaten recently. Sherri Nichols snichols@adobe.com	0	0	1	medical	549	0
9	Article-I.D.: pitt.19407 References: < 1nvo645hal@agate.berkeley.edu> < 19151@pitt.UUCP> < 93087.011308PXF3@psuvm.psu.edu> Sender: news@cs.pitt.edu Reply-To: geb@cs.pitt.edu (Gordon Banks) Organization: Univ. of Pittsburgh Computer Science Lines: 29 In article < 93087.011308PXF3@psuvm.psu.edu> PXF3@psuvm.psu.edu (Paula Ford) writes: How long ago was this? When I said you'd get counselling, I meant if you did it now. Long ago, practices varied and agencies had to gear up to provide the counselling. Good for her. What we don't need is everyone suing community service agencies that provide blood that people need. Testing is not fool proof. The fact that he got AIDS from a transfusion (if he really did) does not mean the Red Cross screwed up. Prior to 1983 or so, there wasn't a good test and a lot of bad blood got through. This wasn't the fault of the Red Cross. When did he get the transfusions? -- -- Gordon Banks N3JXP | "Skepticism is the chastity of the intellect, and geb@cadre.dsl.pitt.edu | it is shameful to surrender it too soon." --	0	0	1	medical	464	0
10	Article-I.D.: jupiter.1993Apr5.131610.17465 Organization: University of New Brunswick Lines: 39 Hartford 1 1 3--6 NY Rangers 1 2 1--4 First period 1, Hartford, Cunneyworth 6 (Janssens, Greig) 12:21. 2, NY Rangers, Graves 34 (Turcotte, Zubov) 18:39. Second period 3, NY Rangers, Kovalev 19 (Turcotte, Graves) 2:12. 4, Hartford, Sanderson 44 (Cassels) pp, 4:54. 5, NY Rangers, Amonte 30 (Andersson, Vanbiesbrouck) pp, 19:13. Third period 6, NY Rangers, M Messier 25 (Amonte, Andersson) 2:26. 7, Hartford, Sanderson 45 (Cassels) sh, 6:23. 8, Hartford, Nylander 6 (Ladouceur) 8:36. 9, Hartford, Verbeek 38 (Zalapski) 17:43. Hartford: 5 Power play: 4-1 Special goals: pp: 1 sh: 1 Total: 2 Scorer G A Pts ---------- --- --- --- Cassels 0 2 2 Cunneyworth 1 0 1 Greig 0 1 1 Janssens 0 1 1 Ladouceur 0 1 1 Nylander 1 0 1 Sanderson 2 0 2 Verbeek 1 0 1 Zalapski 0 1 1 NY Rangers: 4 Power play: 4-1 Scorer G A Pts ---------- --- --- --- Amonte 1 1 2 Andersson 0 2 2 Graves 1 1 2 Kovalev 1 0 1 Messier M 1 0 1 Turcotte 0 2 2 Vanbiesbrouck 0 1 1 Zubov 0 1 1	0	1	0	hockey	278	1

Here is the first 20 rows of the glove CAS table:

The SAS System

Results from table.fetch

Index	.	-0.082418	-0.839884	0.105053	0.022308	-1.474822	-0.209763	0.056989	0.137982	0.372772	0.559319	0.577920	0.144557	-0.701270	-0.404596	0.444879	0.153329	-0.901481
1	of	1.099652	-1.501752	-0.304385	-0.514429	-0.435441	-0.006574	-0.018503	1.640428	1.285993	1.073708	-0.074207	0.509786	-0.330978	-0.262842	0.511022	-0.208204	-0.549805
2	.	-0.505014	-0.964961	0.051956	0.288498	0.229747	-0.329811	0.260266	1.378497	1.059683	1.181037	0.010117	0.503543	0.061392	-0.698514	0.810464	-0.639053	-0.457899
3	the	0.469089	-0.830931	0.119924	-0.923026	-1.033058	0.224467	-0.005171	1.269445	1.377971	0.04187	-0.246549	0.837457	-0.548495	-1.893023	0.55592	-0.628875	-1.038012
4	and	-0.667284	-0.833785	-0.011317	0.305408	-0.586931	-0.180594	0.699553	0.486493	0.194138	0.294581	-0.070622	0.097824	0.525252	-0.030484	0.859129	-0.23034	-0.421106
5	in	0.073027	-1.674842	0.075069	-0.432442	0.574844	-0.106971	-0.967614	-0.221789	1.169087	0.347187	-1.073857	0.357908	-0.174242	-0.007312	0.28522	-0.897391	-1.087383
6	)	0.634824	-1.904375	-0.391289	-0.738393	-0.702658	1.016663	0.523935	0.495943	1.42781	0.524997	0.545411	0.386492	-1.073709	-0.329423	0.486984	0.445128	-0.270627
7	(	0.798141	-1.99449	-0.328878	-0.649862	-0.472035	1.137045	0.505894	0.514905	1.562512	0.627811	0.238156	0.383532	-0.667192	-0.365819	0.752805	0.331056	-0.244839
8	to	0.78572	-0.810246	0.101776	-0.801058	-1.392475	0.16924	-0.263078	-0.015079	0.989012	0.591865	-0.229358	0.987238	-1.421809	-0.782612	1.044341	0.634682	-1.468291
9	with	-0.566415	-0.829788	-0.824924	0.544148	-1.543747	-0.31732	-0.273864	0.068167	1.411421	0.019145	-0.947103	0.756881	-0.505637	0.443736	0.006759	-0.275975	0.666662
10	a	1.507271	-0.407398	1.021648	-0.813827	-1.272684	-0.367007	-0.794286	1.314956	1.057076	0.407871	-0.164403	1.018087	-0.512753	0.557561	0.582248	0.536295	-0.508473
11	patients	0.174284	-1.839482	-0.348263	-0.428796	-1.164599	-0.059588	0.39338	-0.002842	1.304037	-0.147551	-0.440067	0.191422	0.059533	-0.415416	0.378196	-0.277377	-0.30039
12	was	0.723438	-1.099584	0.568463	-0.350693	-1.041354	0.139519	-1.016854	0.571498	1.015211	-0.469769	0.632039	-0.135876	0.254135	-0.601653	0.565271	-0.403599	-1.786361
13	were	0.141054	-1.474184	0.613094	0.18996	-0.91035	0.269496	0.199361	0.329915	0.659171	0.731402	0.462546	-0.705685	-0.089627	-1.220453	0.55314	-0.82889	0.051765
14	for	0.125555	-1.061452	0.52209	-0.224163	-1.14033	-0.625986	0.791364	0.815144	0.040622	0.675383	-0.42299	-0.471853	0.165461	-0.98172	0.652132	-0.059102	-0.895828
15	%	0.475652	-1.616878	0.361061	-0.516853	-0.31318	0.006831	-0.240007	0.439454	1.256686	0.041998	-0.728243	-0.358813	0.356402	-0.129473	0.496482	-0.61118	-1.537968
16	The	0.347747	-0.494994	0.485117	-0.084597	-0.895067	-0.051236	0.103775	0.509738	0.748738	0.214684	0.542988	0.342974	-0.350294	-0.665327	0.61638	-0.376625	-0.990118
17	that	-0.214832	-0.359543	0.635157	0.379445	-0.083595	-0.16214	0.112659	0.369585	1.189184	0.397843	-0.014447	1.188321	-0.680029	-0.213179	-0.133729	0.578246	-0.739475
18	by	0.28878	-0.631891	0.879817	0.745593	-0.750272	0.432009	-0.688888	1.230972	0.097559	0.873638	-0.219702	0.706284	0.26118	0.354966	0.522246	-0.117063	0.522659
19	is	0.531299	0.396706	0.14407	0.348058	-0.214076	-0.616567	-0.200858	0.75753	1.277093	0.252878	0.501842	1.119698	-0.109161	-0.391933	-0.07413	0.233267	-1.551589
20	or	0.388791	-1.326348	-0.031149	0.407569	-0.777309	-1.527304	-0.082104	0.772996	1.179959	0.288621	0.375331	0.206364	-0.309921	0.350062	-0.3142	-0.138743	-0.403793

After you check that your data has been loaded properly, you can build an empty recurrent neural network model and view information about it using actions in the Deep Learning action set:

```
deepLearn.buildModel /                          /*1*/
   modelTable={name="newsRnn",
               replace=TRUE
              }
   type="RNN";
run;

deepLearn.modelInfo /                           /*2*/
   modelTable={name="newsRnn"};
run;
```

1 Uses the buildModel action in the Deep Learning action set to create an empty model for building a recurrent neural network. In this example, the empty model is given the name newsRnn.

2 Uses the modelInfo action in the Deep Learning action set to display model information about the newsRnn model that was created in the previous step.

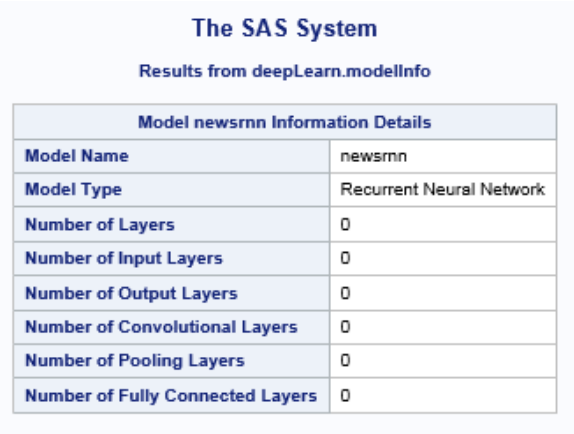
Here is the results from building the empty newsRnn model:

The SAS System

Results from deepLearn.buildModel

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	newsrnn	1	5

Here is the model information for the newsRnn model:



The SAS System

Results from deepLearn.modelInfo

Model newsrnn Information Details	
Model Name	newsrnn
Model Type	Recurrent Neural Network
Number of Layers	0
Number of Input Layers	0
Number of Output Layers	0
Number of Convolutional Layers	0
Number of Pooling Layers	0
Number of Fully Connected Layers	0

Once you have built an empty recurrent neural network model, you can add layers to it using actions in the Deep Learning action set:

```
deepLearn.addLayer /                    /*1*/
   layer={type="INPUT"}
   modelTable={name="newsRnn"}
   name="data";
run;

deepLearn.addLayer /                    /*2*/
   layer={type="recurrent"
          n=50
          act='sigmoid'
          init='xavier'
          rnnType='gru'
         }
   modelTable={name="newsRnn"}
   name="rnn1"
   srcLayers={"data"};
run;

deepLearn.addLayer /                    /*3*/
   layer={type="output"
          act='softmax'
          init='xavier'
         }
   modelTable={name="newsRnn"}
   name="outlayer"
   srcLayers={"rnn1"};
run;
```

1 Uses the addLayer action in the Deep Learning action set to add an input layer to the newsRnn model.

2 Uses the addLayer action in the Deep Learning action set to add a recurrent layer to the newsRnn model.

3 Uses the addLayer action in the Deep Learning action set to add an output layer to the newsRnn model.

Here is the results from adding the input layer to the newsRnn model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	newsrnn	10	5

Here is the results from adding the recurrent layer to the newsRnn model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	newsrnn	23	5

Here is the results from adding the output layer to the newsRnn model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	newsrnn	35	5

Once you have added layers to your model, you can view information about it to see how it has changed:

```
deepLearn.modelInfo /                          /*1*/
   modelTable={name="newsRnn"};
run;

table.fetch /                                  /*2*/
   table={name="newsRnn"};
run;
```

1 Uses the modelInfo action in the Deep Learning action set to display overview information about the newsRnn model.

2 Uses the fetch action in the Table action set to display details information about the newsRnn model.

Here is the overview information about the newsRnn model:

The SAS System

Results from deepLearn.modelInfo

Model newsrnn Information Details	
Model Name	newsrnn
Model Type	Recurrent Neural Network
Number of Layers	3
Number of Input Layers	1
Number of Output Layers	1
Number of Convolutional Layers	0
Number of Pooling Layers	0
Number of Fully Connected Layers	0
Number of Recurrent Layers	1

Here is additional details information about the newsRnn model:

The SAS System

Results from table.fetch

Selected Rows from Table NEWSRNN

Index	_DLKey0_	_DLKey1_	_DLChrVal_	_DLNumVal_	_DLLayerID_
1	data	inputopts.dropout	dropout	0	0
2	outlayer	srclayers.0	rnn1	1	2
3	rnn1	rnnopts.reversed	reversed	0	1
4	outlayer	outputopts.init	Xavier	1	2
5	outlayer	outputopts.no_bias	no_bias	0	2
6	data	inputopts.height	height	0	0
7	rnn1	rnnopts.act	Logistic	2	1
8	outlayer	outputopts.std	std	1	2
9	outlayer	outputopts.mean	mean	0	2
10	data	inputopts.nchannels	nchannels	1	0
11	data	inputopts.width	width	0	0
12	outlayer	outputopts.truncfact	truncfact	0	2
13	data	inputopts.flip	No flipping	1	0
14	outlayer	outputopts.act	Softmax	7	2
15	data	inputopts.std	No Standardization	2	0
16	rnn1	rnnopts.n	n	50	1
17	data	layertype	Input Layer	1	0
18	rnn1	rnnopts.rnntype	Gated Recurrent Unit	2	1
19	data	inputopts.scale	scale	1	0
20	data	inputopts.crop	No cropping	1	0

Once you have reviewed your model, you can train and score it using the data sets that you have previously loaded:

```
deepLearn.dlTrain /                                                    /*1*/
   inputs={{name="text"}}
   modelTable={name="newsRnn"}
   modelWeights={name="newsTrainedWeights",
                 replace=TRUE
                }
```

```
   nThreads=1
   optimizer={algorithm={method="ADAM",
                         lrPolicy='step',
                         gamma=0.5,
                         beta1=0.9,
                         beta2=0.999,
                         learningRate=0.001
                        },
              maxEpochs=10,
              miniBatchSize=1
             }
   seed=54321
   table={name="news_train"}
   texts={{name="text"}}
   target="newsgroup"
   textParms={initInputEmbeddings={importOptions={fileType="AUTO"},
                                   name="glove"
                                  }
             }
   nominal="newsgroup";
run;

deepLearn.dlScore /                                                    /* 2 */
   casOut={name="train_scored",
           replace=TRUE
          }
   initWeights={name="newsTrainedWeights"}
   modelTable={name="newsRnn"}
   table={name="news_test"}
   textParms={initInputEmbeddings={importOptions={fileType="AUTO"},
                                   name="glove"
                                  }
             };
run;
```

1 Uses the dlTrain action in the Deep Learning action set to train the newsRnn model using the news_train and glove CAS data sets.

2 Uses the dlScore action in the Deep Learning action set to score the trained model from the previous step using the news_test CAS data set.

Here is the results from training the newsRnn model using the news_train CAS data set:

The SAS System

Results from deepLearn.dlTrain

Model newsrnn Information Details	
Model Name	newsrnn
Model Type	Recurrent Neural Network
Number of Layers	3
Number of Input Layers	1
Number of Output Layers	1
Number of Convolutional Layers	0
Number of Pooling Layers	0
Number of Fully Connected Layers	0
Number of Recurrent Layers	1
Number of Weight Parameters	22650
Number of Bias Parameters	153
Total Number of Model Parameters	22803
Approximate Memory Cost for Training (MB)	17

Optimization History of Deep Learning Model for NEWS_TRAIN			
Epoch	Learning Rate	Loss	Fit Error
0	0.001	1.1020432533	0.648045
1	0.001	1.0957303891	0.666667
2	0.001	1.0980994556	0.662942
3	0.001	1.0926519925	0.659218
4	0.001	1.0894324922	0.657356
5	0.001	1.0887181059	0.64432
6	0.001	1.0891471641	0.640596
7	0.001	1.0902735317	0.635009
8	0.001	1.0919665558	0.631285
9	0.001	1.0939373925	0.631285

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	newsTrainedWeights	22803	3

Here is the results from scoring the newsRnn model using the news_test CAS data set:

The SAS System

Results from deepLearn.dlScore

Score Information for NEWS_TEST	
Number of Observations Read	60
Number of Observations Used	60
Misclassification Error (%)	65
Loss Error	1.08828

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
casdata	train_scored	60	6

After you have performed scoring, you can view additional information, and export your model:

```
   table.fetch /                                        /* 1 */
      maxRows=20
      table={name="train_scored"};
   run;

   deepLearn.dlExportModel /                            /* 2 */
      casout={name="My_Model"}
      initWeights={name="newsTrainedWeights"}
      modelTable={name="newsRnn"};
   run;

   table.tableInfo /                                    /* 3 */
      name="My_Model";
   run;

   table.tableDetails /                                 /* 4 */
     name="My_Model";
   run;

quit;                                                   /* 5 */
```

1 Uses the fetch action in the Table action set to view information about the train_scored table.

2 Uses the dlExportModel action in the Deep Learning action set to export your finished model. In this example, the model is called My_Model.

3 Uses the tableInfo action in the Table action set to view information about the exported model in the previous step.

4 Uses the tableDetails action in the Table action set to view details information about your exported model.

5 Ends the code to run a CAS action in SAS.

Here is the results from fetching the first 20 rows from the train_scored table:

The SAS System

Results from table.fetch

Selected Rows from Table TRAIN_SCORED

Index	graphics	medical	hockey	_DL_PredName_	_DL_PredP_	_DL_PredLevel_
1	0.3142591119	0.3387875855	0.3469532728	hockey	0.3469532728	2
2	0.1553414762	0.4187365472	0.4259219766	hockey	0.4259219766	2
3	0.3287915587	0.3433358073	0.3278726041	medical	0.3433358073	1
4	0.3624196351	0.377394855	0.2601855695	medical	0.377394855	1
5	0.1284302324	0.6550040245	0.2165657729	medical	0.6550040245	1
6	0.3466865122	0.3774569929	0.2758564949	medical	0.3774569929	1
7	0.3596019745	0.3757745028	0.2646235824	medical	0.3757745028	1
8	0.3497424424	0.3744475245	0.2758100033	medical	0.3744475245	1
9	0.1092060208	0.7563745975	0.1344194561	medical	0.7563745975	1
10	0.155565694	0.4474377334	0.3969965577	medical	0.4474377334	1
11	0.3009450138	0.4705972075	0.2284577489	medical	0.4705972075	1
12	0.3230878115	0.4075587392	0.2693534195	medical	0.4075587392	1
13	0.3743642867	0.345641166	0.2799945772	graphics	0.3743642867	0
14	0.2831509411	0.4917487502	0.2251002789	medical	0.4917487502	1
15	0.1285332292	0.6544519067	0.217014879	medical	0.6544519067	1
16	0.3830802143	0.3568158746	0.2601038814	graphics	0.3830802143	0
17	0.3560261428	0.3756408989	0.268332988	medical	0.3756408989	1
18	0.3543714881	0.3684263527	0.277202189	medical	0.3684263527	1
19	0.1567126811	0.6571804881	0.1861067712	medical	0.6571804881	1
20	0.3430164456	0.3875605464	0.269423008	medical	0.3875605464	1

Here is the results from exporting the model using the dlExportModel action:

The SAS System

Results from deepLearn.dlExportModel

Output CAS Tables	
CAS Library	Name
casdata	My_Model

Here is the overview table information about the exported model:

The SAS System

Results from table.tableInfo

Table Information for Caslib casdata

Table Name	Number of Rows	Number of Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Compressed
MY_MODEL	1	2	utf-8	21Oct2017:02:38:46	21Oct2017:02:38:46	No	Yes	No	No

Here is the details table information about the exported model:

The SAS System

Results from table.tableDetails

Detail Information for My_Model in Caslib casdata.

Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated
ALL	141	141	141	5486006	5462622	141	5491104	0	0	0	0

Example 14: Build a LeNet Model

You can use CASL to build a LeNet Model.

The following example shows how you can use CASL to build a LeNet Model.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";      /*1*/
cas casauto;
caslib _all_ assign;

proc cas;                                              /*2*/
   session casauto;

   deepLearn.buildModel /                              /*3*/
      modelTable={name="LeNet",
                  replace=TRUE
                 }
      type="CNN";
   run;
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Builds an empty convolutional neural network (CNN) model.

Here are the results from building the empty convolutional neural network (CNN) model:

The SAS System

Results from deepLearn.buildModel

Output CAS Tables

CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	1	5

After you build an empty CNN model, you can add layers to it:

```
deepLearn.addLayer /                                   /*1*/
   layer={type="INPUT"
          nchannels=1
```

```
            width=28
            height=28
            scale=0.00392156862745098039
           }
   modelTable={name="LeNet"}
   name="data";
run;

deepLearn.addLayer /                                  /* 2 */
   layer={type="CONVO"
            nFilters=20
            width=5
            height=5
            stride=1
           }
   modelTable={name="LeNet"}
   name="conv1"
   srcLayers={"data"};
run;

deepLearn.addLayer /                                  /* 3 */
   layer={type="POOL"
            width=2
            height=2
            stride=2
           }
   modelTable={name="LeNet"}
   name="pool1"
   srcLayers={"conv1"}
   replace=TRUE;
run;

deepLearn.addLayer /                                  /* 4 */
   layer={type="CONVO"
            nFilters=50
            width=5
            height=5
            stride=1
           }
   modelTable={name="LeNet"}
   name="conv2"
   srcLayers={"pool1"};
run;
```

1 Adds an input layer to the LeNet model.

   Note: This example uses hypothetical values for width and height to illustrate how these parameters can be specified. You need to modify these values to fit your own data and modeling goals.

2 Adds a convolutional layer to the LeNet model with 20 filters, a width of 5, and a height of 5.

3 Adds a pooling layer to the LeNet model with a width of 2 and a height of 2.

4 Adds a convolutional layer to the LeNet model with 50 filters, a width of 5, and a height of 5.

The following adds additional layers:

```
deepLearn.addLayer /                                  /* 1 */
   layer={type="POOL"
```

```
            width=2
            height=2
            stride=2
            pool='max'
           }
   modelTable={name="LeNet"}
   name="pool2"
   srcLayers={"conv2"};
run;

deepLearn.addLayer /                                    /* 2 */
   layer={type="FC"
           n=500
          }
   modelTable={name="LeNet"}
   name="fc1"
   srcLayers={"pool2"};
run;

deepLearn.addLayer /                                    /* 3 */
   layer={type="OUTPUT"
           act='softmax'
          }
   modelTable={name="LeNet"}
   name="outlayer"
   srcLayers={"fc1"};
run;
```

1 Adds a pooling layer to the LeNet model with a width of 2 and a height of 2.

2 Adds a fully connected layer to the LeNet model.

3 Adds an output layer to the LeNet model.

Here are the results for adding the first layer:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	10	5

Here are the results for adding the second layer:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	26	5

Here are the results for adding the third layer:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	35	5

Here are the results for adding the fourth layer:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	51	5

Here are the results for adding the fifth layer:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	60	5

Here are the results for adding the sixth layer:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	71	5

Here are the results for adding the seventh layer:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	83	5

Once layers have been added to the model, you can order them and remove them from the model:

```
fetch/table='LeNet' to=1000 orderby={'_dllayerid_', '_DLKey1_'};          /* 1 */
run;
```

```
deepLearn.removeLayer /                                          /* 2 */
   modelTable={name="LeNet"}
   name="conv2";
run;

deepLearn.removeLayer /                                          /* 3 */
   modelTable={name="LeNet"}
   name="pool2";
run;

fetch/table='LeNet' to=1000 orderby={'_dllayerid_', '_DLKey1_'};  /* 4 */
run;
```

1 Displays table results ordered by the deep learning layer ID and the key value.

2 Once you add layers to a model, you can remove them. This step removes the conv2 layer to illustrate how this can be done.

3 Removes the pool2 layer.

4 Displays table results ordered by the deep learning layer ID and the key value.

Here are the output table results for the first two layers:

The SAS System

Results from table.fetch

Selected Rows from Table LENET

Index	_DLKey0_	_DLKey1_	_DLChrVal_	_DLNumVal_	_DLLayerID_
1	lenet	modeltype	Convolutional Neural Network	2	.
2	data	inputopts.crop	No cropping	1	0
3	data	inputopts.dropout	dropout	0	0
4	data	inputopts.flip	No flipping	1	0
5	data	inputopts.height	height	28	0
6	data	inputopts.nchannels	nchannels	1	0
7	data	inputopts.scale	scale	0.0039215686	0
8	data	inputopts.std	No Standardization	2	0
9	data	inputopts.width	width	28	0
10	data	layertype	Input Layer	1	0
11	conv1	convopts.act	Automatic	0	1
12	conv1	convopts.dropout	dropout	0	1
13	conv1	convopts.height	height	5	1
14	conv1	convopts.init	Xavier	1	1
15	conv1	convopts.initbias	initbias	0	1
16	conv1	convopts.mean	mean	0	1
17	conv1	convopts.nfilters	nfilters	20	1
18	conv1	convopts.no_bias	no_bias	0	1
19	conv1	convopts.std	std	1	1
20	conv1	convopts.stride	stride	1	1
21	conv1	convopts.stride_height	stride_height	1	1
22	conv1	convopts.stride_width	stride_width	1	1
23	conv1	convopts.truncfact	truncfact	0	1
24	conv1	convopts.width	width	5	1
25	conv1	layertype	Convolution Layer	2	1
26	conv1	srclayers.0	data	0	1

Here are the output table results for the next three layers:

27	pool1	layertype	Pooling Layer	3	2
28	pool1	poolingopts.dropout	dropout	0	2
29	pool1	poolingopts.height	height	2	2
30	pool1	poolingopts.poolingtype	Max Pooling	1	2
31	pool1	poolingopts.stride	stride	2	2
32	pool1	poolingopts.stride_height	stride_height	2	2
33	pool1	poolingopts.stride_width	stride_width	2	2
34	pool1	poolingopts.width	width	2	2
35	pool1	srclayers.0	conv1	1	2
36	conv2	convopts.act	Automatic	0	3
37	conv2	convopts.dropout	dropout	0	3
38	conv2	convopts.height	height	5	3
39	conv2	convopts.init	Xavier	1	3
40	conv2	convopts.initbias	initbias	0	3
41	conv2	convopts.mean	mean	0	3
42	conv2	convopts.nfilters	nfilters	50	3
43	conv2	convopts.no_bias	no_bias	0	3
44	conv2	convopts.std	std	1	3
45	conv2	convopts.stride	stride	1	3
46	conv2	convopts.stride_height	stride_height	1	3
47	conv2	convopts.stride_width	stride_width	1	3
48	conv2	convopts.truncfact	truncfact	0	3
49	conv2	convopts.width	width	5	3
50	conv2	layertype	Convolution Layer	2	3
51	conv2	srclayers.0	pool1	2	3
52	pool2	layertype	Pooling Layer	3	4
53	pool2	poolingopts.dropout	dropout	0	4
54	pool2	poolingopts.height	height	2	4
55	pool2	poolingopts.poolingtype	Max Pooling	1	4
56	pool2	poolingopts.stride	stride	2	4
57	pool2	poolingopts.stride_height	stride_height	2	4
58	pool2	poolingopts.stride_width	stride_width	2	4
59	pool2	poolingopts.width	width	2	4
60	pool2	srclayers.0	conv2	3	4

Here are the output table results for the final two layers:

61	fc1	fcopts.act	Automatic	0	5
62	fc1	fcopts.dropout	dropout	0	5
63	fc1	fcopts.init	Xavier	1	5
64	fc1	fcopts.initbias	initbias	0	5
65	fc1	fcopts.mean	mean	0	5
66	fc1	fcopts.n	n	500	5
67	fc1	fcopts.no_bias	no_bias	0	5
68	fc1	fcopts.std	std	1	5
69	fc1	fcopts.truncfact	truncfact	0	5
70	fc1	layertype	Full Connect Layer	4	5
71	fc1	srclayers.0	pool2	4	5
72	outlayer	layertype	Output Layer	5	6
73	outlayer	outputopts.act	Softmax	7	6
74	outlayer	outputopts.error	Automatic	0	6
75	outlayer	outputopts.init	Xavier	1	6
76	outlayer	outputopts.initbias	initbias	0	6
77	outlayer	outputopts.mean	mean	0	6
78	outlayer	outputopts.n	n	0	6
79	outlayer	outputopts.no_bias	no_bias	0	6
80	outlayer	outputopts.std	std	1	6
81	outlayer	outputopts.target_std	target_std	2	6
82	outlayer	outputopts.truncfact	truncfact	0	6
83	outlayer	srclayers.0	fc1	5	6

Here are the results for removing the conv2 layer:

The SAS System

Results from deepLearn.removeLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	66	5

Here are the results for removing the pool2 layer:

The SAS System

Results from deepLearn.removeLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	57	5

Here are the output table results for the first two layers after the conv2 and pool2 layers have been removed:

The SAS System

Results from table.fetch

Selected Rows from Table LENET

Index	_DLKey0_	_DLKey1_	_DLChrVal_	_DLNumVal_	_DLLayerID_
1	lenet	modeltype	Convolutional Neural Network	2	.
2	data	inputopts.crop	No cropping	1	0
3	data	inputopts.dropout	dropout	0	0
4	data	inputopts.flip	No flipping	1	0
5	data	inputopts.height	height	28	0
6	data	inputopts.nchannels	nchannels	1	0
7	data	inputopts.scale	scale	0.0039215686	0
8	data	inputopts.std	No Standardization	2	0
9	data	inputopts.width	width	28	0
10	data	layertype	Input Layer	1	0
11	conv1	convopts.act	Automatic	0	1
12	conv1	convopts.dropout	dropout	0	1
13	conv1	convopts.height	height	5	1
14	conv1	convopts.init	Xavier	1	1
15	conv1	convopts.initbias	initbias	0	1
16	conv1	convopts.mean	mean	0	1
17	conv1	convopts.nfilters	nfilters	20	1
18	conv1	convopts.no_bias	no_bias	0	1
19	conv1	convopts.std	std	1	1
20	conv1	convopts.stride	stride	1	1
21	conv1	convopts.stride_height	stride_height	1	1
22	conv1	convopts.stride_width	stride_width	1	1
23	conv1	convopts.truncfact	truncfact	0	1
24	conv1	convopts.width	width	5	1
25	conv1	layertype	Convolution Layer	2	1
26	conv1	srclayers.0	data	0	1

Here are the output table results for the next three layers:

27	pool1	layertype	Pooling Layer	3	2
28	pool1	poolingopts.dropout	dropout	0	2
29	pool1	poolingopts.height	height	2	2
30	pool1	poolingopts.poolingtype	Max Pooling	1	2
31	pool1	poolingopts.stride	stride	2	2
32	pool1	poolingopts.stride_height	stride_height	2	2
33	pool1	poolingopts.stride_width	stride_width	2	2
34	pool1	poolingopts.width	width	2	2
35	pool1	srclayers.0	conv1	1	2
36	fc1	fcopts.act	Automatic	0	3
37	fc1	fcopts.dropout	dropout	0	3
38	fc1	fcopts.init	Xavier	1	3
39	fc1	fcopts.initbias	initbias	0	3
40	fc1	fcopts.mean	mean	0	3
41	fc1	fcopts.n	n	500	3
42	fc1	fcopts.no_bias	no_bias	0	3
43	fc1	fcopts.std	std	1	3
44	fc1	fcopts.truncfact	truncfact	0	3
45	fc1	layertype	Full Connect Layer	4	3
46	outlayer	layertype	Output Layer	5	4
47	outlayer	outputopts.act	Softmax	7	4
48	outlayer	outputopts.error	Automatic	0	4
49	outlayer	outputopts.init	Xavier	1	4
50	outlayer	outputopts.initbias	initbias	0	4
51	outlayer	outputopts.mean	mean	0	4
52	outlayer	outputopts.n	n	0	4
53	outlayer	outputopts.no_bias	no_bias	0	4
54	outlayer	outputopts.std	std	1	4
55	outlayer	outputopts.target_std	target_std	2	4
56	outlayer	outputopts.truncfact	truncfact	0	4
57	outlayer	srclayers.0	fc1	3	4

You can also add removed layers back into the model:

```
deepLearn.addLayer /                                                    /*1*/
   layer={type="CONVO"
          nFilters=50
          width=5
          height=5
          stride=1
         }
   modelTable={name="LeNet"}
   name="conv2"
   srcLayers={"pool1"};
run;

deepLearn.addLayer /                                                    /*2*/
   layer={type="POOL"
          width=2
          height=2
          stride=2
          pool='max'
         }
   modelTable={name="LeNet"}
```

```
   name="pool2"
   srcLayers={"conv2"};
run;

deepLearn.addLayer /                                                    /*3*/
   layer={type="FC"
          n=500
         }
   modelTable={name="LeNet"}
   name="fc1"
   replace=TRUE
   srcLayers={"pool2"};
run;
```

1 Adds the conv2 layer back into the model.

2 Adds the pool2 layer back into the model.

3 Updates the fully connected layer with the new source layers.

Here are the results for adding the conv2 layer back into the model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	73	5

Here are the results for adding the pool2 layer back into the model:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	82	5

Here are the results for updating the fully connected layer with the new source layers:

The SAS System

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS	lenet	83	5

Finally, you can view table results and summary information about your model:

```
fetch/table='LeNet' to=1000 orderby={'_dllayerid_', '_DLKey1_'};        /*1*/
run;
```

```
   deepLearn.modelInfo /                                                    /*2*/
      modelTable={name="LeNet"};
   run;

quit;                                                                       /*3*/
```

1 Displays table results ordered by the deep learning layer ID and the key value.

2 Displays summary information about the model.

3 Ends the code to run CAS actions in SAS.

After you have configured your model, you can perform training and scoring using the data that you want to use. For more information about training a model, see Train a Deep Learning Model on page 56. For more information about how you can score a table using a Deep Learning model, see Score a Table Using a Deep Learning Model on page 54.

Here are the output table results for the first two layers after the conv2 and pool2 layers have been added back and the fully connected layer has been updated:

The SAS System

Results from table.fetch

Selected Rows from Table LENET

Index	_DLKey0_	_DLKey1_	_DLChrVal_	_DLNumVal_	_DLLayerID_
1	lenet	modeltype	Convolutional Neural Network	2	.
2	data	inputopts.crop	No cropping	1	0
3	data	inputopts.dropout	dropout	0	0
4	data	inputopts.flip	No flipping	1	0
5	data	inputopts.height	height	28	0
6	data	inputopts.nchannels	nchannels	1	0
7	data	inputopts.scale	scale	0.0039215686	0
8	data	inputopts.std	No Standardization	2	0
9	data	inputopts.width	width	28	0
10	data	layertype	Input Layer	1	0
11	conv1	convopts.act	Automatic	0	1
12	conv1	convopts.dropout	dropout	0	1
13	conv1	convopts.height	height	5	1
14	conv1	convopts.init	Xavier	1	1
15	conv1	convopts.initbias	initbias	0	1
16	conv1	convopts.mean	mean	0	1
17	conv1	convopts.nfilters	nfilters	20	1
18	conv1	convopts.no_bias	no_bias	0	1
19	conv1	convopts.std	std	1	1
20	conv1	convopts.stride	stride	1	1
21	conv1	convopts.stride_height	stride_height	1	1
22	conv1	convopts.stride_width	stride_width	1	1
23	conv1	convopts.truncfact	truncfact	0	1
24	conv1	convopts.width	width	5	1
25	conv1	layertype	Convolution Layer	2	1
26	conv1	srclayers.0	data	0	1

Here are the output table results for the next three layers:

27	pool1	layertype	Pooling Layer	3	2
28	pool1	poolingopts.dropout	dropout	0	2
29	pool1	poolingopts.height	height	2	2
30	pool1	poolingopts.poolingtype	Max Pooling	1	2
31	pool1	poolingopts.stride	stride	2	2
32	pool1	poolingopts.stride_height	stride_height	2	2
33	pool1	poolingopts.stride_width	stride_width	2	2
34	pool1	poolingopts.width	width	2	2
35	pool1	srclayers.0	conv1	1	2
36	conv2	convopts.act	Automatic	0	3
37	conv2	convopts.dropout	dropout	0	3
38	conv2	convopts.height	height	5	3
39	conv2	convopts.init	Xavier	1	3
40	conv2	convopts.initbias	initbias	0	3
41	conv2	convopts.mean	mean	0	3
42	conv2	convopts.nfilters	nfilters	50	3
43	conv2	convopts.no_bias	no_bias	0	3
44	conv2	convopts.std	std	1	3
45	conv2	convopts.stride	stride	1	3
46	conv2	convopts.stride_height	stride_height	1	3
47	conv2	convopts.stride_width	stride_width	1	3
48	conv2	convopts.truncfact	truncfact	0	3
49	conv2	convopts.width	width	5	3
50	conv2	layertype	Convolution Layer	2	3
51	conv2	srclayers.0	pool1	2	3
52	pool2	layertype	Pooling Layer	3	4
53	pool2	poolingopts.dropout	dropout	0	4
54	pool2	poolingopts.height	height	2	4
55	pool2	poolingopts.poolingtype	Max Pooling	1	4
56	pool2	poolingopts.stride	stride	2	4
57	pool2	poolingopts.stride_height	stride_height	2	4
58	pool2	poolingopts.stride_width	stride_width	2	4
59	pool2	poolingopts.width	width	2	4
60	pool2	srclayers.0	conv2	3	4

Here are the output table results for the final two layers:

61	fc1	fcopts.act	Automatic	0	5
62	fc1	fcopts.dropout	dropout	0	5
63	fc1	fcopts.init	Xavier	1	5
64	fc1	fcopts.initbias	initbias	0	5
65	fc1	fcopts.mean	mean	0	5
66	fc1	fcopts.n	n	500	5
67	fc1	fcopts.no_bias	no_bias	0	5
68	fc1	fcopts.std	std	1	5
69	fc1	fcopts.truncfact	truncfact	0	5
70	fc1	layertype	Full Connect Layer	4	5
71	fc1	srclayers.0	pool2	4	5
72	outlayer	layertype	Output Layer	5	6
73	outlayer	outputopts.act	Softmax	7	6
74	outlayer	outputopts.error	Automatic	0	6
75	outlayer	outputopts.init	Xavier	1	6
76	outlayer	outputopts.initbias	initbias	0	6
77	outlayer	outputopts.mean	mean	0	6
78	outlayer	outputopts.n	n	0	6
79	outlayer	outputopts.no_bias	no_bias	0	6
80	outlayer	outputopts.std	std	1	6
81	outlayer	outputopts.target_std	target_std	2	6
82	outlayer	outputopts.truncfact	truncfact	0	6
83	outlayer	srclayers.0	fc1	5	6

Here are the model information results:

The SAS System

Results from deepLearn.modelInfo

Model lenet Information Details	
Model Name	lenet
Model Type	Convolutional Neural Network
Number of Layers	7
Number of Input Layers	1
Number of Output Layers	1
Number of Convolutional Layers	2
Number of Pooling Layers	2
Number of Fully Connected Layers	1
Total Number of Model Parameters	83

Details

Overview of the Deep Learning Actions

The Deep Learning action set provides actions for modeling and scoring with deep learning networks.

For an example that shows how you can use the actions in the Deep Learning action set together, see Build a Deep Learning Model with Tabular Data on page 68.

Note: You need to have a SAS Visual Data Mining and Machine Learning license to use the actions in the Deep Learning action set.

About the Deep Learning Actions

Add a Layer

The addLayer action adds a layer to a deep learning model.

When specifying the source layer for the `srcLayers` parameter, note that only directed acyclic graphs are supported.

For an example that shows how you can add a layer to a deep learning model, see Add a Layer to a Deep Learning Model on page 43.

Build a New Model

The buildModel action builds an empty deep learning model.

For an example that shows how you can build an empty deep learning model, see Build a Deep Learning Model on page 45.

Export a Model

The dlExportModel action exports a deep learning model.

For an example that shows how you can export a deep learning model, see Export a Deep Learning Model on page 46.

Import Model Weights

The dlImportModelWeights action imports model weights from an external source.

For an example that shows how you can import model weights, see Import Model Weights on page 48.

Assign Target Label Information

The dlLabelTarget action assigns target label information.

For an example that shows how you can assign target label information, see Assign Target Label Information on page 52.

Score a Table

The dlScore action scores a table using a deep learning model.

The `layerOut` parameter specifies the settings for an output table that includes layer output values. By default, all layers are included. Note that this consumes a large amount of memory. You can filter the list with the `layers` parameter.

For an example that shows how you can score a table using a deep learning model, see Score a Table Using a Deep Learning Model on page 54.

Train a Model

The dlTrain action trains a deep learning model.

When training your model, you can use the `optimizer` parameter to specify the settings for the optimization algorithm, optimization mode, and other settings such as a seed, the maximum number of epochs, and so on.

For example, the following table illustrates how the fit error in a convolutional neural network can improve with each epoch.

§ ModelInfo

	Descr	Value
0	Model Name	tnet
1	Model Type	Convolutional Neural Network
2	Number of Layers	9
3	Number of Input Layers	1
4	Number of Output Layers	1
5	Number of Convolutional Layers	3
6	Number of Pooling Layers	3
7	Number of Fully Connected Layers	1
8	Number of Weight Parameters	1163616
9	Number of Bias Parameters	779
10	Approximate Memory Cost for Training (MB)	127

§ OptIterHistory

	Epoch	LearningRate	Loss	FitError
0	0.0	0.001	2.382182	0.669346
1	1.0	0.001	0.519747	0.167300
2	2.0	0.001	0.193625	0.061454
3	3.0	0.001	0.122883	0.037596
4	4.0	0.001	0.086291	0.026406
5	5.0	0.001	0.076109	0.021456
6	6.0	0.001	0.078402	0.021670
7	7.0	0.001	0.059615	0.016039
8	8.0	0.001	0.059523	0.015130
9	9.0	0.001	0.058977	0.014877
10	10.0	0.001	0.043318	0.011399
11	11.0	0.001	0.046885	0.011156
12	12.0	0.001	0.054555	0.012617
13	13.0	0.001	0.050729	0.010204

In this example, the `maxEpochs` option inside the `optimizer` parameter was set to a value of 14. The `maxEpochs` option specifies the maximum number of epochs. For SGD with a single-machine server or a session that uses one worker on a distributed server, one epoch is reached when the action passes through the data one time. For a session that uses more than one worker, one epoch is reached when all the workers exchange the weights with the controller one time.

When using asynchronous SGD with the `snapshotFreq` parameter to specify the frequency for generating snapshots of the neural weights, this slows down the training speed because the action synchronizes all the weights before writing them out in a weight table.

For an example that shows how you can train a deep learning model, see Train a Deep Learning Model on page 56.

Tune a Model

The dlTune action tunes hyper-parameters for a deep learning model. It enables you to tune optimization parameters for the dlTrain action using a hyperband tuning method.

When the `gpu` parameter is specified, the dlTune action uses graphical processing unit hardware to accelerate the dlTune action. You can identify GPU devices with 0–based 64–bit integers. For example, `devices={0, 2}` requests that the first and the third GPU device be used. When this option is not specified, the action uses up to the first nThreads= GPU devices. For example, if there are two GPU devices available, then `nThreads=4` and `gpu=1` implies the action uses GPU devices 0 and 1. If `nThreads=1` and `gpu=1` is specified, then it means that only GPU device 0 is used even if there are multiple GPU devices available.

The following output illustrates how tuning can lead to decreased fit error with subsequent iterations:

	Iteration	PointsRemaining	BestPoint	BestFitError
0	0.0	18.0	7.0	0.6932
1	1.0	13.0	10.0	0.6778
2	2.0	10.0	13.0	0.6528
3	3.0	7.0	24.0	0.6149
4	4.0	5.0	14.0	0.5828
5	5.0	4.0	7.0	0.6016
6	6.0	3.0	14.0	0.5983
7	7.0	3.0	14.0	0.5909
8	8.0	3.0	7.0	0.6028
9	9.0	3.0	14.0	0.5834

The following output contains additional tuning statistics:

	Learning rate	Beta 1	Beta 2	Iterations	ValidFitError
0	0.0054954095	0.97114	0.90746	20	0.5834
1	0.0015135612	0.83582	0.96318	20	0.6427
2	0.0724435896	0.82786	0.9791	20	0.6586
3	0.0199526232	0.87562	0.81194	14	0.6409
4	0.0028840317	0.95522	0.97114	12	0.6527
5	0.0380189344	0.96318	0.99502	10	0.6667
6	0.0104712825	0.93134	0.98706	10	0.6857
7	0.0007943282	0.91542	0.8597	8	0.6919
8	0.0004168693	0.8995	0.88358	8	0.7028
9	0.0001148153	0.98706	0.8199	8	0.7106
10	0.0002187761	0.8597	0.95522	6	0.7367
11	0.0000602559	0.81194	0.8995	6	0.7949
12	0.0000316228	0.88358	0.82786	6	0.8137
13	8.7096314E-6	0.99502	0.89154	4	0.8527
14	0.0000165959	0.80398	0.91542	4	0.8582
15	4.5708794E-6	0.9791	0.87562	4	0.8843
16	2.3988314E-6	0.90746	0.83582	4	0.8885
17	1.2589257E-6	0.85174	0.92338	4	0.89
18	6.6069356E-7	0.8199	0.80398	2	0.8915
19	5.0118672E-8	0.84378	0.85174	2	0.8923
20	3.467369E-7	0.92338	0.84378	2	0.8923
21	1.8197009E-7	0.94726	0.93134	2	0.8923
22	9.5499168E-8	0.9393	0.9393	2	0.8923
23	1.3803827E-8	0.89154	0.86766	2	0.8924
24	2.6302651E-8	0.86766	0.94726	2	0.8924

For an example that shows how you can tune hyper-parameters for a deep learning model, see Tune a Deep Learning Model on page 58.

Display Model Information

The modelInfo action shows model information.

For an example that shows how you can display model information, see Display Model Information on page 61.

Remove a Layer

The removeLayer action removes a layer from a deep learning model.

When specifying the name of the layer to remove from the model using the `name` parameter, note that only directed acyclic graphs are supported. After removing a layer, the graph connections need to be restored.

For an example that shows how you can remove a layer from a deep learning model, see Remove a Layer from a Deep Learning Model on page 63.

Minimum Batch Size

The minimum batch size is the maximum number of observations across all workers. Often, the minimum batch size is the product of the number of threads you are using and the value of parameter `minibatchsize`. However, if you are working with a smaller data set, such as a table with five rows, five threads, and a `minibatchsize` value of 1, your actual minimum batch size can be just one if all five rows are assigned to the first thread, and the remaining four threads get no data.

Deep Neural Network Action Set: Examples

Example 1: Generate DATA Scoring Code from a Deep Neural Network Model

You can use CASL to generate DATA scoring code from a deep neural network model using the dnnCode action.

The following example shows how you can use CASL to generate DATA scoring code from a deep neural network model using the dnnCode action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /*1*/
cas casauto;
caslib _all_ assign;

proc cas;                                             /*2*/
   session casauto;

   table.loadTable result=r /                         /*3*/
      caslib="Library_name_to_use"
      path="Path_to_my_file"
      casOut={name="CAS_file_name", replace=true};
   run;

   <Additional steps for a model table>               /*4*/

   deepNeural.dnnCode  /                              /*5*/
      initWeights={name="IW_table_name"}
      modelTable={name="Model_table_name"};
   run;

quit;                                                 /*6*/
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Loads the source data into a table.

   Note: You need to modify the `caslib`, `path`, and `name` parameters to fit the library name, path, and table names that you want to use.

4 After you connect to a CAS session, and load the table that you want to use, you might need to perform additional steps to load or create a model table.

5 Calls the dnnCode action in the Deep Neural Network action set using the in-memory table that contains the model weights and the model table that you want to use.

Note: You need to modify the `name` values for the `initWeights` and `modelTable` parameters to fit the model weights table and the model table that you want to use.

6 Ends the code to run a CAS action in SAS.

Example 2: Export a Neural Network Model

You can use CASL to export a deep fully connected neural network model using the dnnExportModel action.

The following example shows how you can use CASL to export a deep fully connected neural network model using the dnnExportModel action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                             /* 2 */
   session casauto;

   table.loadTable result=r /                         /* 3 */
      caslib="Library_name_to_use"
      path="Path_to_my_file"
      casOut={name="CAS_file_name", replace=true};
   run;

   <Additional steps for a model table>               /* 4 */

   deepNeural.dnnExportModel  /                       /* 5 */
      casout={name="Out_table_name"}
      initWeights={name="IW_table_name"}
      modelTable={name="Model_table_name"};
   run;

quit;                                                 /* 6 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Loads the source data into a table.

Note: You need to modify the `caslib`, `path`, and `name` parameters to fit the library name, path, and table names that you want to use.

4 After you connect to a CAS session, and load the table that you want to use, you might need to perform additional steps to load or create a model table.

5 Calls the dnnExportModel action in the Deep Neural Network action set using the table to store the generated aStore model, the in-memory table that contains the model weights, and the model table that you want to use.

Note: You need to modify the `name` values for the `casout`, `initWeights`, and `modelTable` parameters to fit the aStore model table, the model weights table, and the model table that you want to use.

6 Ends the code to run a CAS action in SAS.

Example 3: Score a Table with a Deep Neural Network Model

You can use CASL to score a table with a deep neural network model using the dnnScore action.

The following example shows how you can use CASL to score a table with a deep neural network model using the dnnScore action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /*1*/
cas casauto;
caslib _all_ assign;

proc cas;                                             /*2*/
   session casauto;

   table.loadTable result=r /                         /*3*/
      caslib="Library_name_to_use"
      path="Path_to_my_file"
      casOut={name="CAS_file_name", replace=true};
   run;

   <Additional steps for a model table>               /*4*/

   deepNeural.dnnScore  /                             /*5*/
      initWeights={name="IW_table_name"}
      modelTable={name="Model_table_name"}
      table={name="Table_name"};
   run;

quit;                                                 /*6*/
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Loads the source data into a table.

Note: You need to modify the `caslib`, `path`, and `name` parameters to fit the library name, path, and table names that you want to use.

4 After you connect to a CAS session, and load the table that you want to use, you might need to perform additional steps to load or create a model table.

5 Calls the dnnScore action in the Deep Neural Network action set using the in-memory table that contains the model weights, the model table, and the input table that you want to use.

Note: You need to modify the `name` values for the `initWeights`, `modelTable`, and `table` parameters to fit the model weights table, the model table, and the input table that you want to use.

6 Ends the code to run a CAS action in SAS.

Example 4: Train a Deep Neural Network

You can use CASL to train a deep neural network using the dnnTrain action.

The following example shows how you can use CASL to train a deep neural network using the dnnTrain action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                             /* 2 */
   session casauto;

   table.loadTable result=r /                         /* 3 */
      caslib="Library_name_to_use"
      path="Path_to_my_file"
      casOut={name="CAS_file_name", replace=true};
   run;

   deepNeural.dnnTrain  /                             /* 4 */
      inputs={{name="Variable_name"}}
      modelOut={name="MO_table_name"}
      modelWeights={name="MW_table_name"}
      table={name="CAS_file_name"};
   run;

quit;                                                 /* 5 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Loads the source data into a table.

   Note: You need to modify the `caslib`, `path`, and `name` parameters to fit the library name, path, and table names that you want to use.

4 Calls the dnnTrain action in the Deep Neural Network action set using the input variables to use in the analysis, the in-memory table that is used to store the model, the in-memory table that contains the model weights, and the input table that you want to use.

   Note: You need to modify the `name` values for the `inputs`, `modelOut`, `modelWeights`, and `table` parameters to fit the variables, model table, model weights table, and the input table that you want to use.

5 Ends the code to run a CAS action in SAS.

Details

Overview of the Deep Neural Network Actions

The Deep Neural Network action set provides actions for working with deep neural network models.

Note: You need to have a SAS Visual Data Mining and Machine Learning license to use the actions in the Deep Neural Network action set.

About the Deep Neural Network Actions

Generate DATA Scoring Code

The dnnCode action generates DATA scoring code from a deep neural network model.

Export a Model

The dnnExportModel action exports a neural network model.

Score a Table

The dnnScore action scores a table with a deep neural network model.

Train a Deep Neural Network

The dnnTrain action trains a deep neural network.

Recurrent Neural Network Action Set: Examples

Example 1: Score a Table Using a Recurrent Neural Network Model

You can use CASL to score a table with a recurrent neural network model using the rnnScore action.

The following example shows how you can use CASL to score a table with a recurrent neural network model using the rnnScore action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /*1*/
cas casauto;
caslib _all_ assign;

proc cas;                                             /*2*/
   session casauto;

   table.loadTable result=r /                         /*3*/
      caslib="Library_name_to_use"
      path="Path_to_my_file"
      casOut={name="CAS_file_name", replace=true};
   run;

   <Additional steps for a model table>               /*4*/

   deepRnn.rnnScore  /                                /*5*/
      initWeights={name="IW_table_name"}
      modelTable={name="Model_table_name"}
      table={name="Table_name"};
   run;

quit;                                                 /*6*/
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Loads the source data into a table.

   Note: You need to modify the `caslib`, `path`, and `name` parameters to fit the library name, path, and table names that you want to use.

4 After you connect to a CAS session, and load the table that you want to use, you might need to perform additional steps to load or create a model table.

5 Calls the rnnScore action in the Recurrent Neural Network action set using the in-memory table that contains the model weights, the model table, and the input table that you want to use.

Note: You need to modify the `name` values for the `initWeights`, `modelTable`, and `table` parameters to fit the model weights table, the model table, and the input table that you want to use.

6 Ends the code to run a CAS action in SAS.

Example 2: Train a Recurrent Neural Network

You can use CASL to train a recurrent neural network using the rnnTrain action.

The following example shows how you can use CASL to train a recurrent neural network using the rnnTrain action.

Note: Before running the following code, you need to provide a CAS port number and a CAS host name for the `casport` and `cashost` parameters. The values for these parameters that are provided below are just an example.

```
options casport=5570 cashost="cloud.example.com";     /* 1 */
cas casauto;
caslib _all_ assign;

proc cas;                                             /* 2 */
   session casauto;

   table.loadTable result=r /                         /* 3 */
      caslib="Library_name_to_use"
      path="Path_to_my_file"
      casOut={name="CAS_file_name", replace=true};
   run;

   deepRnn.rnnTrain  /                                /* 4 */
      hiddens={5, 3}
      inputs={{name="Variable_name"}}
      modelOut={name="MO_table_name"}
      modelWeights={name="MW_table_name"}
      table={name="CAS_file_name"}
      target="T_variable_name";
   run;

quit;                                                 /* 5 */
```

1 Specifies the CAS port number and CAS host name to use. If you have already connected to a CAS session, skip to step 2.

2 Begins the code to run a CAS action in SAS.

3 Loads the source data into a table.

Note: You need to modify the `caslib`, `path`, and `name` parameters to fit the library name, path, and table names that you want to use.

4 Calls the rnnTrain action in the Recurrent Neural Network action set using the number of hidden neurons for each hidden layer in the model, the input variables to use in the analysis, the in-memory table that is used to store the model, the in-memory table that contains the model weights, the input table, and the target variable that you want to use. In the above code, two hidden layers are specified. One has 5 hidden neurons, and the other has 3 hidden neurons.

Note: You need to modify the `name` values for the `inputs`, `modelOut`, `modelWeights`, and `table` parameters to fit the variables, model table, model weights table, and the input table that you want to use. You also need to modify the value for the `target` parameter to fit the target variable that you want to use.

5 Ends the code to run a CAS action in SAS.

Details

Overview of the Recurrent Neural Network Actions

The Recurrent Neural Network action set provides actions for working with recurrent neural network models.

Note: You need to have a SAS Visual Data Mining and Machine Learning license to use the actions in the Recurrent Neural Network action set.

About the Recurrent Neural Network Actions

Score a Table

The rnnScore action scores a table using a recurrent neural network model.

Train a Recurrent Neural Network

The rnnTrain action trains a recurrent neural network.