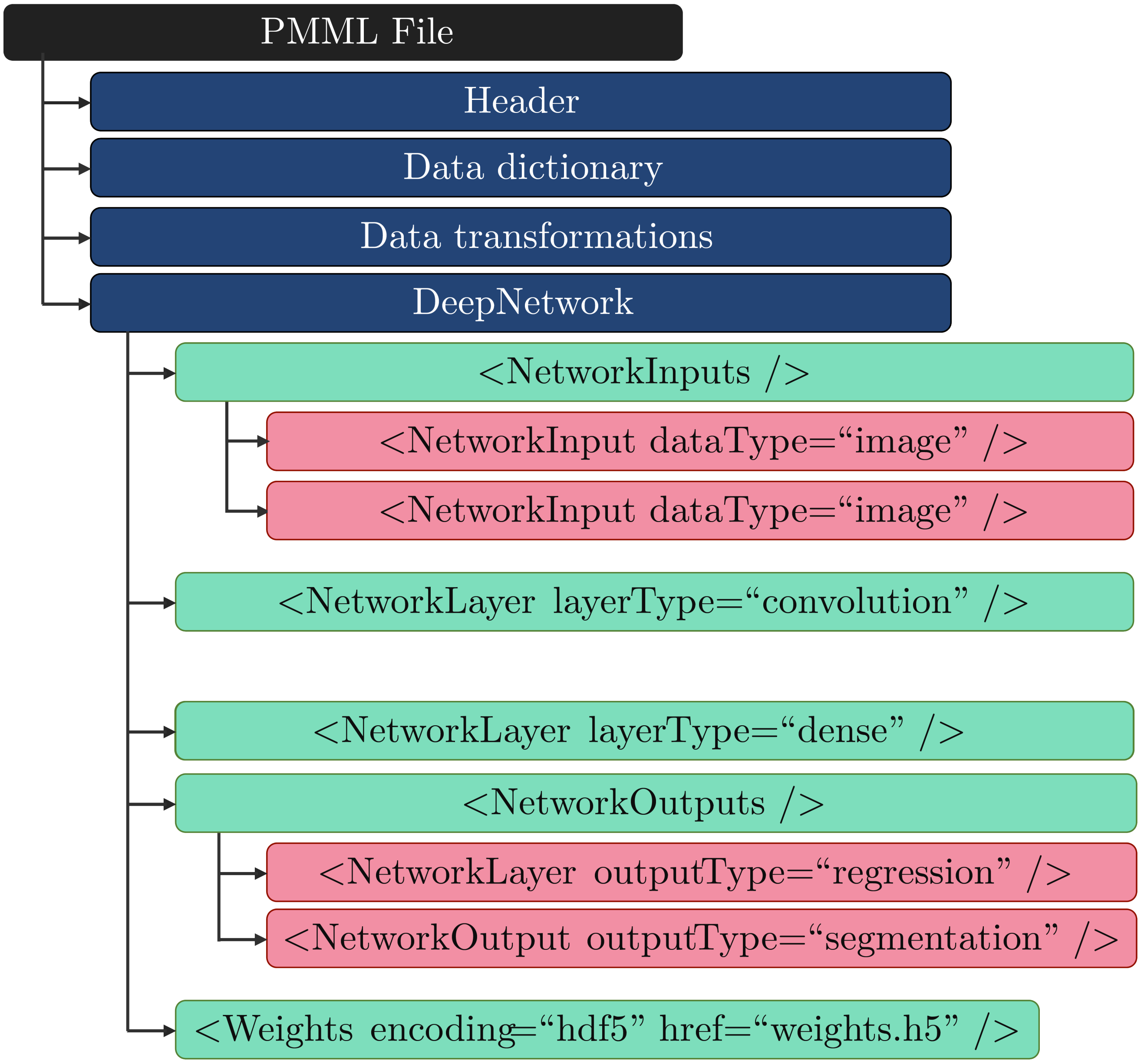
PMML 5.0 - Convolutional Neural Network

Convolutional neural networks, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. The PMML specification for CNN focuses on the application of CNNs to image data. In a CNN, pixels from each image are converted to a featurized representation through a series of mathematical operations. Input images are represented as an order 3 tensor with height `H`, width `W`, and depth `D`. This representation is modified by several hidden layers until the desired output is achieved. Several layer types are common to most modern CNNs, including convolution, pooling and dense layer types. The PMML specification describes each how the overall network and each layer is represented.


PMML File
Header
Data dictionary
Data transformations
DeepNetwork
<NetworkInputs />
<NetworkInput dataType="image" />
<NetworkInput dataType="image" />
<NetworkLayer layerType="convolution" />
<NetworkLayer layerType="dense" />
<NetworkOutputs />
<NetworkLayer outputType="regression" />
<NetworkOutput outputType="segmentation" />
<Weights encoding="hdf5" href="weights.h5" />

DeepNetwork Element

A CNN model is represented by a *DeepNetwork* element, which contains all the necessary information to fully characterize the model. The *DeepNetwork* element can have three types of child elements:

- **NetworkInputs** element defines inputs to the neural network
- **NetworkLayer** elements define the hidden layers in the neural network
- **NetworkOutputs** element defines the outputs of `the neural network.

A *DeepNetwork* element must have at least one *NetworkInput* and at least one *NetworkOuput*. The *DeepNetwork* element must contain one or more *NetworkLayer* elements that describe individual nodes in the dataflow graph. The XML schema for the DeepNetwork element is shown below:

```
<xs:element name="DeepNetwork">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="MiningSchema"/>
      <xs:element name="NetworkOutputs" />
      <xs:element maxOccurs="unbounded" name="NetworkLayer" />
      <xs:element name="Weights">
        <xs:complexType>
          <xs:attribute name="encoding" type="xs:string" use="required" />
          <xs:attribute name="href" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="modelName" type="xs:string" use="required" />
    <xs:attribute name="functionName" type="xs:string" use="required" />
    <xs:attribute name="numberOfLayers" type="xs:nonNegativeInteger"
use="required" />
  </xs:complexType>
</xs:element>
```

Neural Network Layers

A deep neural network is represented as a directed acyclic graph. Each node in the graph represents a neural network layer. Graph edges describe the connections between neural network layers. The *NetworkLayer* element is used to define the node in the proposed DeepNetwork PMML extension. Similarly, the *NetworkInputs* element is used to describe the input to the neural network. The *layerName* attribute of each *NetworkLayer* and *NetworkInputs* elements uniquely identifies each neural network layer. It is possible to connect layers by specifying the inputs to each layer. Specifically, a

NetworkLayer can have a child *InboundNodes* element which defines inputs to the layer. If the *InboundNodes* child is not present, then it is assumed that the layer does not have any inputs.

```
<xs:element maxOccurs="unbounded" name="NetworkLayer">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" name="InboundNodes" />
      <xs:element minOccurs="0" name="TargetShape" />
      <xs:element minOccurs="0" name="PoolSize" />
      <xs:element minOccurs="0" name="Size" />
      <xs:element minOccurs="0" name="Strides">
      <xs:element minOccurs="0" name="ConvolutionalKernel">
      <xs:element minOccurs="0" name="Padding">
      <xs:element minOccurs="0" name="InputSize">
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Convolution Layer

The `convolution` layer convolves a convolutional kernel with the input tensor. The convolution layer is represented using a *NetworkLayer* element with *layerType* set to Conv2D. A *NetworkLayer* with the Conv2D layer type must contain a *ConvolutionalKernel* child element, which describes the properties of the convolutional kernel. The cardinality of the convolutional tensor must be equal to that of the input tensor. The size of the convolutional kernel is governed by the parameter *KernelSize* child element, and the stride is governed by the parameter *KernelStrides.* An activation function can be optionally applied to the output of this layer. An example of a convolutional layer is provided below:

```
<NetworkLayer activation="relu" layerType="Conv2D" name="block2_conv1"
padding="same" use_bias="True">
  <InboundNodes>
    <Array n="1" type="string">block1_pool</Array>
  </InboundNodes>
  <ConvolutionalKernel channels="128">
    <DilationRate>
      <Array n="2" type="int">1 1</Array>
    </DilationRate>
    <KernelSize>
      <Array n="2" type="int">3 3</Array>
    </KernelSize>
    <KernelStride>
      <Array n="2" type="int">1 1</Array>
    </KernelStride>
  </ConvolutionalKernel>
</NetworkLayer>
```

Dense Layer

A `dense` layer is represented using a *NetworkLayer* element with *layerType* set to Dense. An activation function can be optionally applied to the output of this layer. This layer is parameterized by weights **W** and bias **b**. An example of a dense layer is provided below:

```
<NetworkLayer activation="relu" channels="4096" layerType="Dense" name="fc1">
  <InboundNodes>
    <Array n="1" type="string">flatten</Array>
  </InboundNodes>
</NetworkLayer>
```

Merge Layer

A `merge` layer takes two or more tensors of equal dimensions and combines them using an elementwise operator. A merge layer is represented using a *NetworkLayer* element with the *layerType* attribute set to Merge. The *operator* attribute is used to specify the operator used to combine tensor values. Allowable operator types are addition, subtraction, multiplication, and division. An example of a dense layer is provided below:

```
<NetworkLayer layerType="Merge" name="add_1" operator="add">
  <InboundNodes>
    <Array n="2" type="string">bn2a_branch2c bn2a_branch1</Array>
  </InboundNodes>
</NetworkLayer>
```

Pooling Layer

`Pooling` layers apply a pooling operation over a single tensor. A max pooling layer is represented using a *NetworkLayer* element with the *layerType* attribute set to MaxPooling2D. An average pooling layer is represented using a *NetworkLayer* element with the *layerType* attribute set to AveragePooling2D. The width of the pooling kernel is governed by the *PoolSize* child element*,* and the stride is governed by the parameter *PoolStrides* child element*.*

```
<NetworkLayer layerType="MaxPooling2D" name="pool1">
    <InboundNodes>
        <Array n="1" type="string">zero_padding2d_2</Array>
    </InboundNodes>
    <PoolSize>
      <Array n="2" type="int">3 3</Array>
    </PoolSize>
    <Strides>
      <Array n="2" type="int">2 2</Array>
    </Strides>
</NetworkLayer>
```

Global Pooling Layer

`Global pooling` layers apply a pooling operation across all spatial dimensions of the input tensor. A global max pooling layer is represented using a *NetworkLayer* element with the *layerType* attribute set to GlobalMaxPooling2D. A global average pooling layer is represented using a *NetworkLayer* element with the *layerType* attribute set to GlobalAveragePooling2D. Both of these global pooling layers return a tensor that has size *(batch_size, channels)*.

```
<NetworkLayer layerType="GlobalAveragePooling2D" name="avg_pool">
  <InboundNodes>
    <Array n="1" type="string">activation_49</Array>
  </InboundNodes>
</NetworkLayer>
```

Depthwise Convolution Layer

The `depthwise convolution layer` convolves a convolutional filter with the input, keeping each channel separate. In the regular convolution layer, convolution is performed over multiple input channels. The depth of the filter is equal to the number of input channels, allowing values across multiple channels to be combined to form the output. Depthwise convolutions keep each channel separate - hence the name *depthwise*. A depthwise convolution layer is represented using a *NetworkLayer* element with the *layerType* attribute set to "DepthwiseConv2D".

```
<NetworkLayer activation="linear" depth_multiplier="1"
layerType="DepthwiseConv2D" name="conv_dw_1" padding="same" use_bias="False">
    <InboundNodes>
      <Array n="1" type="string">conv1_relu</Array>
    </InboundNodes>
    <ConvolutionalKernel>
      <KernelSize>
        <Array n="2" type="int">3 3</Array>
      </KernelSize>
      <KernelStride>
        <Array n="2" type="int">1 1</Array>
      </KernelStride>
    </ConvolutionalKernel>
</NetworkLayer>
```

Batch Normalization Layer

The `batch normalization` (BN) layer aims to generate an output tensor with a constant mean and variance. BN applies a linear transformation between input and output based on the distribution of inputs during the training process. A BN layer is represented using a *NetworkLayer* element with the *layerType* attribute set to "BatchNormalization".

```
<NetworkLayer layerType="BatchNormalization" axis="-1" center="True"
epsilon="0.001" momentum="0.99" name="conv_dw_2_bn">
  <InboundNodes>
    <Array n="1" type="string">conv_dw_2</Array>
  </InboundNodes>
</NetworkLayer>
```

Activation Layer

The `activation` layer applies an activation function to each element in the input tensor. An activation layer is represented using a *NetworkLayer* element with the *layerType* attribute set to Activation. The activation function can be any one of linear, relu, sigmoid, tanh, elu, or softmax. The attribute *threshold* allows the activation function to be offset horizontally.

```
<NetworkLayer layerType="Activation" activation="relu" max_value="6.0"
name="conv1_relu" negative_slope="0.0" threshold="0.0">
    <InboundNodes>
        <Array n="1" type="string">conv1_bn</Array>
    </InboundNodes>
</NetworkLayer>
```

Padding Layer

A `padding` layer pads the spatial dimensions of a tensor with a constant value, often zero. This operation is commonly used to increase the size of oddly shaped layers, to allow dimension reduction in subsequent layers. In the proposed PMML format, a padding layer is represented using a *NetworkLayer* element with the *layerType* attribute set to Padding2D.

```
<NetworkLayer layerType="Padding2D" name="conv_pad_2">
  <InboundNodes>
    <Array n="1" type="string">conv_pw_1_relu</Array>
  </InboundNodes>
  <Padding>
    <Array n="4" type="int">0 1 0 1</Array>
  </Padding>
</NetworkLayer>
```

Reshape Layer

A `reshape` layer reshapes the input tensor. The number of values in the input tensor must equal the number of values in the output tensor. The first dimension is not reshaped as this is commonly the batch dimension. In the proposed PMML format, a padding layer is represented using a *NetworkLayer* element with the *layerType* attribute set to Reshape. The flatten layer is a variant of the reshape layer, that flattens the input tensor such that the output size is where is the number of values in the input tensor. In the proposed PMML format, a flatten layer is represented using a *NetworkLayer* element with the *layerType* attribute set to Flatten.

```
<NetworkLayer layerType="Reshape" name="reshape_1">
  <InboundNodes>
    <Array n="1" type="string">global_average_pooling2d_1</Array>
  </InboundNodes>
  <TargetShape>
    <Array n="3" type="int">1 1 1024</Array>
  </TargetShape>
</NetworkLayer>
```

Transposed Convolution

`Transposed convolutions`, also called deconvolutions, arise from the desire to use a transformation going in the opposite direction of a normal convolution, for example, to increase the spatial dimensions of a tensor. They are commonly used in CNN decoders, which progressively increase the spatial size of the input tensor. In the PMML CNN format, a transposed convolution layer is represented using a *NetworkLayer* element with the *layerType* attribute set to TransposedConv2D.

Neural Network Output

CNN's are commonly used for image classification, segmentation, regression, and object localization. PMML supports all of these output types. The *NetworkOutputs* element is used to define the outputs of a deep neural network. Each output is defined using the *NetworkOutput* element.

Classification Output

```
<NetworkOutput name="classification_1">
    <DerivedField optype="categorical" field="concat_1" dataType="string">
        <DiscretizeClassification classes="class" />
    <DerivedField />
<NetworkOutput />
```

Regression Output

```
<NetworkOutput name="regression_1">
    <FieldRef field="dense_3" dataType="double" />
<NetworkOutput />
```

Localization Output

```
<NetworkOutput name="localization_1">
    <FieldRef field="dense_2" dataType="tensor" />
<NetworkOutput />
```

Segmentation Output

```
<NetworkOutput name="segmentation_1">
    <DerivedField optype="categorical" field="concat_1" dataType="tensor">
        <DiscretizeSegmentation classes="class" field="pool_2" />
    </DerivedField />
<NetworkOutput />
```

PMML can be used to represent CNN classification models. Classification models approximate a mapping function (f) from input variables (X) to a discrete output variable (y). The output variables are often called labels or categories. The mapping function predicts the class or category for a given observation. For example, an image can be classified as belonging to one of two classes: “cat” or “dog”. It is common for classification models to predict a continuous value as the probability of a given example belonging to each output class. The probabilities can be interpreted as the likelihood or confidence of a given example belonging to each class. A predicted probability can be converted into a class value by selecting the class label that has the highest probability. The proposed extension introduces a *DiscretizeClassification* element that defines this transformation. Specifically, *DiscretizeClassification* describes a transformation that takes an input tensor of class *likelihoods* and outputs a string describing the most probable class.

PMML can also be used to represent CNN regression models. Formally, regression models approximate a mapping function (f) from input variables (X) to a continuous output variable (Y). A continuous output variable is a real-value, such as an integer or floating-point value, or a tensor of continuous values. The existing *FieldRef* PMML element is used to define regression models. If a *FieldRef* is contained in a *NetworkOuput* then it returns a copy of any specified tensor in the neural network. If the *FieldRef* has a double datatype, then it converts a single element tensors to a single double value.

PMML can be used to represent CNN segmentation models. Semantic segmentation is one of the fundamental tasks in computer vision. In semantic segmentation, the goal is to classify each pixel of the image in a specific category. Formally, semantic segmentation models approximate a mapping function (*f*) from an input image (X) to a tensor of object classes (Y). Most modern neural network architectures learn a mapping between the input image and a tensor that describes the class likelihood for each pixel, where   is the number of predefined classes. The final step is to select the class with the largest likelihood for each pixel. The proposed PMML extension introduces a *DiscretizeSegmentation* element that defines this transformation. Specifically, *DiscretizeSegmentation* describes a transformation that takes an input tensor and outputs a tensor containing the most likely pixel classes.

Example

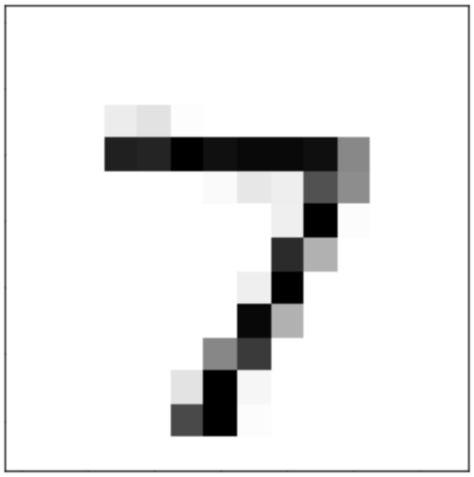
In this example, PMML is used to represent a CNN model to classify hand-written digits. The input of the model is a grayscale image with a size of 14 x 14 x 1. The model contains two convolutional layers, each with a single 3x3x1 convolutional filter. The model also contains a *MaxPooling* layer to reduce the spatial size of the feature representation and two *Dense* layers that predict the hand-written digits. The output of the model is a string from the set {"zero", "one", ... "nine"} that matches the number in the input image. The PMML file for the trained model is shown below:

```
<?xml version='1.0' encoding='UTF-8'?>
<PMML version="5.0" xmlns="http://www.dmg.org/PMML-5_0">
  <Header copyright="Copyright (c) 2020 NIST" description="Simple model to detect
hand-written digits">
    <Timestamp>2020-40-15 10:40:56</Timestamp>
  </Header>
  <DataDictionary numberOfFields="2">
    <DataField channels="1" dataType="tensor" height="14" name="I"
optype="categorical" width="14"/>
    <DataField dataType="string" name="class" optype="categorical">
      <Value value="Zero"/>
      <Value value="One"/>
      <Value value="Two"/>
      <Value value="Three"/>
      <Value value="Four"/>
      <Value value="Five"/>
      <Value value="Six"/>
      <Value value="Seven"/>
      <Value value="Eight"/>
      <Value value="Nine"/>
    </DataField>
  </DataDictionary>
  <DeepNetwork modelName="Deep Neural Network" functionName="classification"
numberOfLayers="9">
    <MiningSchema>
      <MiningField name="image" usageType="active"/>
      <MiningField name="class" usageType="predicted"/>
    </MiningSchema>
    <Outputs>
      <OutputField dataType="string" feature="topClass"/>
    </Outputs>
    <NetworkLayer layerType="InputLayer" name="input_2">
      <InputSize>
        <Array n="3" type="int">14 14 1</Array>
      </InputSize>
    </NetworkLayer>
    <NetworkLayer activation="relu" layerType="Conv2D" name="conv2d_2"
padding="valid" use_bias="True">
      <InboundNodes>
        <Array n="1" type="string">input_2</Array>
      </InboundNodes>
      <ConvolutionalKernel channels="1">
        <DilationRate>
          <Array n="2" type="int">1 1</Array>
        </DilationRate>
        <KernelSize>
          <Array n="2" type="int">3 3</Array>
        </KernelSize>
        <KernelStride>
          <Array n="2" type="int">1 1</Array>
        </KernelStride>
      </ConvolutionalKernel>
    </NetworkLayer>
    <NetworkLayer activation="relu" layerType="Conv2D" name="conv2d_3"
padding="valid" use_bias="True">
      <InboundNodes>
        <Array n="1" type="string">conv2d_2</Array>
      </InboundNodes>
      <ConvolutionalKernel channels="1">
        <DilationRate>
          <Array n="2" type="int">1 1</Array>
        </DilationRate>
        <KernelSize>
          <Array n="2" type="int">3 3</Array>
        </KernelSize>
        <KernelStride>
          <Array n="2" type="int">1 1</Array>
        </KernelStride>
      </ConvolutionalKernel>
    </NetworkLayer>
    <NetworkLayer layerType="MaxPooling2D" name="max_pooling2d_1">
      <InboundNodes>
        <Array n="1" type="string">conv2d_3</Array>
      </InboundNodes>
      <PoolSize>
        <Array n="2" type="int">2 2</Array>
      </PoolSize>
      <Strides>
        <Array n="2" type="int">2 2</Array>
      </Strides>
    </NetworkLayer>
    <NetworkLayer layerType="Flatten" name="flatten_1">
      <InboundNodes>
        <Array n="1" type="string">conv2d_3</Array>
      </InboundNodes>
    </NetworkLayer>
    <NetworkLayer activation="relu" channels="16" layerType="Dense"
name="dense_2">
      <InboundNodes>
        <Array n="1" type="string">flatten_1</Array>
      </InboundNodes>
    </NetworkLayer>
    <NetworkLayer activation="softmax" channels="10" layerType="Dense"
name="dense_3">
      <InboundNodes>
        <Array n="1" type="string">dense_2</Array>
      </InboundNodes>
    </NetworkLayer>
    <Weights encoding="hdf5" href="small_model.h5"/>
  </DeepNetwork>
</PMML>
```

The weights are stored using the HDF5 format, which is a binary key-value format. HDF5 is not human readable but can be converted to a human-readable format like JSON for display purposes. The contents of the HDF5 file are shown in JSON below:

```
{
  "conv2d_2": {
    "bias:0": [0.16033408045768738],
    "kernel:0": `<Tensor shape=(3, 3, 1, 1)>`
  },
  "conv2d_3": {
    "bias:0": [-0.14729286730289 46],
    "kernel:0": `<Tensor shape=(3, 3, 1, 1)>`
  },
  "dense_2": {
    "bias:0": `<Tensor shape=(16,)>`,
    "kernel:0": `<Tensor shape=(25, 16)>`
  },
  "dense_3": {
    "bias:0": `<Tensor shape=(10,)>`,
    "kernel:0": `<Tensor shape=(16, 10)>`
  },
  "dropout_2": {},
  "dropout_3": {},
  "flatten_1": {},
  "input_2": {},
  "max_pooling2d_1": {}
}
```

Large tensors have been replaced by `<Tensor>` for display purposes. The model is evaluated against the single-channel grayscale image shown below:



This image can be expressed as a Matrix:

$$x_1 = \begin{bmatrix}
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & -0.00 & 0.08 & 0.12 & 0.01 & -0.03 & -0.02 & -0.02 & -0.02 & -0.01 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & -0.01 & 0.87 & 0.85 & 1.05 & 0.94 & 0.96 & 0.96 & 0.94 & 0.47 & -0.01 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & -0.04 & -0.03 & -0.07 & 0.02 & 0.10 & 0.07 & 0.68 & 0.45 & -0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.06 & 1.15 & 0.01 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & -0.04 & 0.83 & 0.31 & -0.01 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.06 & 1.03 & -0.03 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & -0.08 & 0.97 & 0.31 & -0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & -0.01 & 0.47 & 0.77 & -0.02 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.11 & 1.12 & 0.04 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & -0.02 & 0.71 & 1.02 & 0.01 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00
\end{bmatrix}$$

All numbers are rounded to two decimal places for visual clarity. The unrounded values are used for calculations. The first layer in the DeepNetwork file is a Convolution layer. The convolution kernel (W_1) and bias (b_1) are loaded from the weights file. The output of the first layer (a_1) can be calculated as follows:

$$W_1 = \begin{bmatrix} -0.82 & -0.87 & -0.97 \\ -0.02 & 0.10 & 0.17 \\ 0.55 & 0.22 & 0.08 \end{bmatrix}$$

$$b_1 = \begin{bmatrix} 0.16 \end{bmatrix}$$

$$\begin{aligned} a_1 &= \mathrm{ReLU}(W_1 * x_1 + b_1) \\ &= \begin{bmatrix}
0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.16 \\
0.16 & 0.17 & 0.19 & 0.23 & 0.23 & 0.16 & 0.14 & 0.14 & 0.14 & 0.14 & 0.15 & 0.16 \\
0.16 & 0.24 & 0.44 & 0.93 & 0.93 & 1.02 & 0.96 & 0.98 & 0.94 & 0.79 & 0.42 & 0.16 \\
0.16 & 0.23 & 0.20 & 0.19 & 0.29 & 0.40 & 0.50 & 0.58 & 0.58 & 0.69 & 0.40 & 0.16 \\
0.17 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & 0.16 \\
0.16 & 0.20 & 0.22 & 0.29 & 0.23 & 0.11 & 0.06 & -0.00 & -0.00 & -0.00 & -0.00 & 0.16 \\
0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.16 & 0.33 & -0.00 & -0.00 & -0.00 & 0.15 & 0.16 \\
0.16 & 0.16 & 0.16 & 0.16 & 0.15 & 0.27 & -0.00 & -0.00 & -0.00 & -0.00 & 0.16 & 0.16 \\
0.16 & 0.16 & 0.16 & 0.16 & 0.18 & 0.42 & -0.00 & -0.00 & -0.00 & 0.18 & 0.16 & 0.16 \\
0.16 & 0.16 & 0.16 & 0.17 & 0.43 & -0.00 & -0.00 & -0.00 & -0.00 & 0.16 & 0.16 & 0.16 \\
0.16 & 0.16 & 0.16 & 0.24 & 0.14 & -0.00 & -0.00 & -0.00 & 0.18 & 0.16 & 0.16 & 0.16 \\
0.16 & 0.16 & 0.16 & 0.17 & -0.00 & -0.00 & -0.00 & 0.13 & 0.16 & 0.16 & 0.16 & 0.16
\end{bmatrix} \end{aligned}$$

where "*" indicates the convolution operator and "+" is an element-wise addition. The output of the second convolution layer (a_2) is calculated as:

$$W_2 = \begin{bmatrix} -0.47 & -0.60 & -0.78 \\ 0.19 & 0.35 & -0.11 \\ 0.53 & 0.40 & 0.56 \end{bmatrix}$$

$$b_2 = \begin{bmatrix} -0.15 \end{bmatrix}$$

$$a_2 = \mathrm{ReLU}(W_2 * a_1 + b_2)$$
$$= \begin{bmatrix} 0.05 & 0.46 & 0.78 & 1.10 & 1.09 & 1.10 & 1.05 & 0.95 & 0.66 & 0.29 \\ -0.00 & -0.00 & 0.10 & 0.32 & 0.56 & 0.75 & 0.84 & 0.94 & 0.80 & 0.46 \\ -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 \\ -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 \\ 0.08 & 0.18 & 0.21 & 0.21 & 0.26 & 0.11 & 0.04 & -0.00 & -0.00 & -0.00 \\ -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & 0.06 & -0.00 & -0.00 & -0.00 & -0.00 \\ -0.00 & -0.00 & -0.00 & 0.01 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 \\ -0.00 & -0.00 & 0.02 & -0.00 & 0.02 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 \\ -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 \\ -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 & -0.00 \end{bmatrix}$$

The next layer in the CNN is a *MaxPooling2D* layer with a 2x2 kernel size. The stride of the pooling kernel is 2. The MaxPooling2D layer does not have any learanable parameters stored in the weights file. Applying the pooling operation gives the output of the third layer (a_3).

$$a_3 = MaxPool(a_2)$$
$$= \begin{bmatrix} 0.46 & 1.10 & 1.10 & 1.05 & 0.80 \\ -0.00 & -0.00 & -0.00 & -0.00 & -0.00 \\ 0.18 & 0.21 & 0.26 & 0.04 & -0.00 \\ -0.00 & 0.02 & 0.02 & -0.00 & -0.00 \\ -0.00 & -0.00 & -0.00 & -0.00 & -0.00 \end{bmatrix}$$

The output is flattened into a single vector using a *Flatten* layer:

$$a_4 = Flatten(a_3)$$

$$= \begin{bmatrix} 0.46 \\ 1.10 \\ 1.10 \\ 1.05 \\ 0.80 \\ -0.00 \\ -0.00 \\ -0.00 \\ -0.00 \\ -0.00 \\ 0.18 \\ 0.21 \\ 0.26 \\ 0.04 \\ -0.00 \\ -0.00 \\ 0.02 \\ 0.02 \\ -0.00 \\ -0.00 \\ -0.00 \\ -0.00 \\ -0.00 \\ -0.00 \\ -0.00 \end{bmatrix}$$

The next layer in the CNN is a *Dense* layer. The weight matrix (W_5) and the bias (b_5) are loaded from the weights file. The output of the dense layer (a_5) can be calulated as follows:

$$W_5 = \begin{bmatrix} 0.12 & -0.02 & 0.38 & 0.61 & 0.24 & 0.46 & -0.24 & 0.15 & -0.14 & 0.09 & 0.20 & 0.69 & 0.03 & -0.34 & -0.09 & -0.08 \\ -0.05 & 0.25 & 0.29 & 0.32 & 0.06 & 0.30 & 0.04 & 0.05 & 0.01 & -0.18 & 0.23 & -0.00 & -0.13 & -0.07 & -0.06 & -0.07 \\ -0.06 & -0.03 & 0.17 & 0.07 & 0.00 & 0.12 & 0.00 & 0.01 & 0.00 & 0.45 & -0.07 & 0.17 & -0.08 & -0.06 & -0.03 & -0.03 \\ -0.03 & 0.15 & 0.16 & 0.05 & 0.06 & 0.11 & -0.00 & -0.45 & -0.00 & -0.08 & 0.04 & -0.00 & -0.13 & -0.07 & -0.02 & -0.02 \\ -0.12 & 0.23 & 0.20 & 0.10 & -0.05 & 0.15 & -0.00 & 0.01 & -0.01 & -0.06 & -0.05 & 0.19 & -0.09 & -0.01 & -0.05 & -0.07 \\ 0.25 & 0.46 & 0.07 & 0.51 & 0.01 & 0.27 & 0.16 & 0.15 & -0.07 & -0.08 & 0.19 & 0.29 & -0.18 & 0.07 & -0.12 & -0.03 \\ 0.45 & -0.34 & -0.06 & 0.25 & 0.12 & -0.08 & -0.21 & -0.38 & 0.15 & -0.19 & -0.14 & 0.43 & 0.16 & -0.21 & 0.14 & 0.05 \\ 0.13 & -0.49 & -0.01 & -0.04 & -0.11 & 0.03 & 0.27 & -0.38 & 0.33 & -0.02 & -0.11 & 0.07 & 0.14 & -0.08 & -0.05 & 0.04 \\ -0.00 & -0.21 & -0.02 & -0.02 & -0.12 & -0.01 & 0.49 & -0.54 & 0.44 & -0.08 & 0.56 & -0.04 & -0.15 & 0.10 & -0.03 & 0.09 \\ -0.08 & -0.06 & -0.04 & -0.08 & -0.28 & -0.19 & 0.55 & -0.35 & 0.69 & 0.01 & 0.77 & -0.05 & -0.44 & -0.26 & -0.02 & 0.09 \\ 0.91 & -0.39 & -0.13 & 0.92 & 0.03 & 0.29 & -0.01 & -0.40 & -0.22 & -0.41 & -0.21 & 0.07 & 0.47 & 0.18 & 0.39 & -0.07 \\ 0.27 & -0.15 & -0.04 & 0.38 & 0.20 & 0.24 & 0.25 & -0.07 & -0.22 & -0.10 & -0.07 & -0.18 & 0.22 & 0.17 & 0.11 & -0.18 \\ -0.11 & 0.05 & -0.00 & 0.05 & 0.02 & 0.15 & 0.19 & 0.02 & 0.04 & 0.20 & -0.06 & -0.01 & 0.14 & 0.30 & -0.04 & -0.26 \\ -0.11 & 0.00 & -0.00 & -0.01 & -0.26 & -0.09 & 0.16 & -0.22 & 0.15 & -0.03 & -0.16 & -0.02 & 0.09 & 0.38 & 0.04 & 0.21 \\ 0.37 & -0.20 & -0.10 & 0.00 & -0.71 & -0.45 & 0.36 & -0.66 & 0.29 & -0.32 & -0.05 & 0.02 & 0.21 & 0.30 & 0.03 & 0.11 \\ 0.54 & -0.31 & 0.09 & 0.80 & -0.13 & -0.02 & 0.20 & -0.32 & 0.50 & 0.07 & -0.18 & 0.38 & -0.23 & -0.32 & 0.54 & 0.31 \\ 0.16 & 0.22 & 0.37 & 0.40 & -0.14 & -0.14 & -0.03 & -0.05 & 0.03 & -0.02 & -0.09 & 0.55 & -0.32 & 0.09 & 0.41 & 0.31 \\ 0.00 & 0.17 & 0.18 & 0.10 & -0.26 & -0.34 & -0.07 & -0.12 & -0.03 & -0.26 & -0.10 & 0.01 & -0.25 & 0.24 & 0.49 & 0.39 \\ 0.09 & 0.10 & -0.10 & 0.01 & -0.61 & -0.41 & -0.18 & -0.55 & -0.04 & -0.19 & -0.24 & -0.02 & 0.04 & 0.45 & 0.45 & 0.32 \\ 0.59 & -0.19 & -0.21 & 0.09 & 0.07 & -0.28 & -0.08 & -0.24 & -0.27 & -0.26 & -0.13 & -0.02 & 0.59 & 0.73 & -0.31 & -0.48 \\ -0.24 & -0.14 & -0.12 & 0.26 & -0.00 & -0.09 & 0.56 & -0.06 & 0.13 & 0.20 & 0.20 & 0.10 & -0.11 & -0.31 & 0.44 & 0.42 \\ 0.18 & -0.27 & 0.40 & 0.03 & -0.14 & -0.09 & 0.54 & -0.20 & 0.38 & 0.14 & 0.54 & 0.44 & -0.21 & -0.57 & 0.41 & 0.50 \\ 0.05 & -0.24 & 0.35 & -0.06 & -0.58 & -0.24 & 0.00 & -0.26 & 0.51 & -0.22 & 0.17 & 0.50 & 0.22 & -0.10 & 0.06 & 0.70 \\ 0.02 & -0.08 & -0.36 & 0.27 & -0.62 & -0.36 & 0.37 & -0.18 & 0.11 & -0.12 & 0.30 & -0.12 & 0.27 & -0.22 & 0.16 & 0.54 \\ -0.21 & 0.03 & 0.04 & 0.19 & -0.21 & -0.29 & -0.14 & 0.14 & -0.29 & -0.02 & 0.15 & -0.10 & 0.43 & 0.48 & -0.27 & 0.10 \end{bmatrix}$$

$$b_5 = \begin{bmatrix} -0.01 \\ 0.00 \\ -0.10 \\ -0.03 \\ 0.34 \\ 0.00 \\ -0.00 \\ 0.43 \\ 0.00 \\ -0.41 \\ -0.00 \\ -0.11 \\ 0.00 \\ 0.00 \\ -0.32 \\ -0.00 \end{bmatrix}$$

$$a_5 = \mathrm{ReLU}(W_5 a_4 + b_5)$$

$$= \begin{bmatrix} -0.00 \\ 0.51 \\ 0.88 \\ 1.08 \\ 0.57 \\ 1.04 \\ 0.03 \\ 0.00 \\ -0.00 \\ -0.00 \\ 0.20 \\ 0.52 \\ -0.00 \\ -0.00 \\ -0.00 \\ -0.00 \end{bmatrix}$$

The first *Dense* layer is followed by a second *Dense* layer. The weight matrix (W_6) and the bias (b_6) are loaded from the weights file. The output of the second *Dense* layer (a_6) can be calulated as follows:

$$W_6 = \begin{bmatrix} -0.15 & -0.94 & 0.41 & 0.36 & -0.02 & 0.21 & -0.16 & -0.03 & 0.15 & -0.34 \\ 0.40 & -0.64 & -0.31 & 0.01 & -0.10 & 0.14 & -0.61 & 0.24 & -0.03 & 0.12 \\ 0.11 & -0.80 & -0.36 & 0.10 & -0.10 & 0.10 & -0.87 & 0.15 & -0.00 & 0.12 \\ -0.02 & -1.13 & 0.10 & 0.10 & -0.08 & 0.05 & -0.96 & 0.10 & -0.04 & -0.02 \\ -0.26 & 0.53 & 0.21 & -0.31 & 0.20 & -0.64 & -0.45 & 0.51 & -0.07 & 0.33 \\ -0.03 & -0.67 & 0.12 & -0.14 & 0.13 & -0.03 & -0.80 & 0.38 & 0.15 & 0.26 \\ -0.58 & -2.16 & 0.09 & 0.00 & 0.07 & 0.17 & 0.24 & -0.18 & -0.01 & 0.05 \\ 0.11 & 0.45 & -0.36 & -0.87 & -0.04 & -0.69 & -0.08 & 0.33 & -0.36 & 0.09 \\ -0.52 & -1.73 & -0.00 & 0.20 & 0.08 & 0.25 & 0.27 & -0.28 & 0.09 & 0.04 \\ 0.01 & -0.47 & -0.12 & -0.28 & -0.12 & 0.12 & -0.50 & 0.23 & 0.07 & 0.25 \\ -0.45 & -1.45 & -0.35 & -0.12 & -0.08 & 0.05 & 0.15 & 0.05 & -0.06 & -0.02 \\ 0.08 & -0.94 & -0.01 & 0.22 & 0.02 & 0.17 & -0.81 & 0.12 & -0.01 & 0.06 \\ -0.49 & -1.36 & 0.48 & -0.07 & -0.22 & -0.31 & -0.07 & -0.35 & -0.07 & -0.49 \\ 0.24 & -0.81 & 0.34 & 0.04 & -0.14 & 0.11 & 0.20 & -0.50 & 0.02 & -0.27 \\ 0.28 & -0.63 & 0.15 & 0.31 & -0.70 & 0.27 & -0.13 & -0.55 & -0.10 & -0.39 \\ 0.17 & -1.49 & -0.05 & 0.25 & -0.42 & 0.21 & 0.10 & -0.88 & -0.12 & -0.26 \end{bmatrix}$$

$$b_6 = \begin{bmatrix} -0.08 \\ 1.23 \\ -0.41 \\ -0.46 \\ 0.16 \\ -0.64 \\ 0.30 \\ -0.32 \\ 0.05 \\ -0.13 \end{bmatrix}$$

$$\begin{aligned} a_6 &= \text{Softmax}(W_6 a_5 + b_6) \\ &= \begin{bmatrix} 0.11 \\ 0.01 \\ 0.06 \\ 0.07 \\ 0.14 \\ 0.05 \\ 0.00 \\ 0.25 \\ 0.12 \\ 0.18 \end{bmatrix} \end{aligned}$$

The output of this layer (a_6) is the score for each class. The PMML file specifies that the model should return the "topClass". The top class can be calculated by applying an argmax operation to a_6:

$$\begin{aligned}\text{index} &= \arg\max(a_6) \\ &= \arg\max \begin{bmatrix} -0.08 \\ 1.23 \\ -0.41 \\ -0.46 \\ 0.16 \\ -0.64 \\ 0.30 \\ -0.32 \\ 0.05 \\ -0.13 \end{bmatrix} \\ &= 7\end{aligned}$$

Here, the number 7 refers to the element 7 in the DataDictionary. Element 7 in the data dictionary is "seven", so the model returns "seven"