Proceedings of the ASME 2019 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2019
August 18 - 21, 2019, Anaheim, California, USA

IDETC2019-97095

A STANDARDIZED REPRESENTATION OF CONVOLUTIONAL NEURAL NETWORKS FOR RELIABLE DEPLOYMENT OF MACHINE LEARNING MODELS IN THE MANUFACTURING INDUSTRY

Max Ferguson
Seongwoon Jeong
Kincho H. Law
Engineering Informatics Group
Stanford University
Stanford, California, 94305

Svetlana Levitan
IBM Cognitive Applications
IBM
Chicago, Illinois, 60606

Anantha Narayanan
Viterbi School of Engineering
University of Southern California
Los Angeles, California, 90007

Rainer Burkhardt
Artificial Intelligence R&D
SoftwareAG
San Diego, California, 92130

Tridivesh Jena
Artificial Intelligence R&D
Zementis*
San Diego, California, 92130

Yung-Tsun Tina Lee
Systems Integration Division
National Institute of Standards and Technology
Gaithersburg, Maryland, 20899

ABSTRACT

The use of deep convolutional neural networks is becoming increasingly popular in the engineering and manufacturing sectors. However, managing the distribution of trained models is still a difficult task, partially due to the limitations of standardized methods for neural network representation. This paper seeks to address this issue by proposing a standardized format for convolutional neural networks, based on the Predictive Model Markup Language (PMML). A number of pre-trained ImageNet models are converted to the proposed PMML format to demonstrate the flexibility and utility of this format. These models are then fine-tuned to detect casting defects in Xray images. Finally, a scoring engine is developed to evaluate new input images against models in the proposed format. The utility of the proposed format and scoring engine is demonstrated by benchmarking the performance of the defect-detection models on a range of different computation platforms. The scoring engine and trained models are made available at `https://github.com/maxkferg/python-pmml`

*Zementis was acquired by SoftwareAG on 1/1/2017 .

INTRODUCTION

Convolutional neural networks (CNNs) are finding numerous real-world use cases in the manufacturing domain [1]. Recent research has demonstrated that convolutional neural networks can obtain state-of-the-art performance on tasks like casting defect detection [2], anomaly detection in fibrous materials [3], and classification of waste recycling [4]. Recent progress in transfer learning has greatly reduced training dataset requirements, allowing powerful models to be trained with relatively small datasets [2]. However, sharing and deploying trained models still remains a difficult and error-prone task. Modern CNNs often have hundreds of layers and millions of parameters, making the distribution and deployment of these models a challenging task. In current practice, models are often saved using a serialization format specific to the machine learning framework used to train the model. In most cases, this method provides a reliable method of saving trained models, but it greatly hinders interoperability between different machine learning frameworks. In this paper, we seek to address this issue by developing a standardized representation of CNNs, based on the Predictive Model Markup

Language (PMML).

With the development and adoption of the Predictive Model Markup Language, it is now much easier to train and evaluate predictive models on separate computers. PMML is an XML-based language that enables the definition and sharing of predictive models between applications [5]. It provides a clean and standardized interface between the software tools that produce predictive models, such as statistical or data mining systems, and the consumers of models, such as applications that depend upon embedded analytics [6]. With PMML it is easy to train a model with a statistical package such as R, and save the model in a standardized format for use in a real-world application.

For deployment, predictive models are normally evaluated by a scoring engine. A scoring engine is a piece of software specifically designed to load a model in a standardized format, and use it to evaluate new observations or data points. Scoring engines are responsible for executing the mathematical operations that transform model inputs into model outputs. To promote interoperability, scoring engines are normally written in languages such as Python, C++ or Java, which are supported by most embedded systems and computing environments. Therefore, it is feasible to run the same scoring engine on a development computer, a cloud server, or an embedded device, without modifying the scoring engine code. The development of standards-compliant scoring engines allows PMML models to be reliably evaluated on a range of devices.

Neural Networks: An artificial neural network (ANN) is an interconnected group of nodes, akin to the vast network of neurons in a brain. The ANN itself is not an algorithm, but rather a framework for many different machine learning algorithms to work together and process complex data inputs [7]. These systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules. For example, an ANN could be trained to identify manufacturing defects by exposing it to example images that have been manually labeled as "defective" or "not defective". Such neural networks are generally trained with little prior knowledge about materials or manufacturing defects. Instead, neural networks automatically generate identifying characteristics from the learning material that they process. In common ANN implementations, the signal at a connection between artificial neurons is a real number, and the output of each artificial neuron is computed by some non-linear function of the sum of its inputs. The connections between artificial neurons are called "edges". Artificial neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Typically, artificial neurons are aggregated into layers. Different layers may perform different kinds of transformations on their inputs.

Convolutional Neural Networks: The development of CNNs has led to vast improvements in many image processing tasks. In a CNN, pixels from each image are converted to a featurized representation through series of mathematical operations. Images can be represented as an order 3 tensor $I \in \mathbb{R}^{H \times W \times D}$ with height H, width W, and D color channels [8]. The input sequentially goes through a number of processing steps, commonly referred to as layers. Each layer i can be viewed as an arbitrary transformation $x_{i+1} = f(x_i; \theta_i)$ with inputs x_i, outputs x_{i+1}, and parameters θ_i. By combining multiple layers it is possible to develop a complex nonlinear function which can map high-dimensional data (such as images) to useful outputs (such as classification labels) [39]. More formally, a CNN can be thought of as the composition of number of functions:

$$f_x = f_N(...(f_2(f_1(x_1;\theta_1);\theta_2)...);\theta_N), \tag{1}$$

where x_1 is the input to the CNN and f_x is the output. There are several layer types which are common to most modern CNNs, including convolution layers, pooling layers and batch normalization layers. A convolution layer is a function $f_i(x_i; \theta_i)$ that convolves one or more parameterized kernels with the input tensor, x_i. Suppose the input x_i is an order 3 tensor with size $H_i \times W_i \times D_i$. A convolution kernel is also an order 3 tensor with size $H \times W \times D_i$. The kernel is convolved with the input by taking the dot product of the kernel with the input at each spatial location in the input. By convolving certain types of kernels with the input image, it is possible to obtain meaningful outputs, such as the image gradients. In most modern CNN architectures, the first few convolutional layers extract features like edges and textures. Convolutional layers deeper in the network can extract features that span a greater spatial area of the image, such as object shapes.

Dataflow Graphs: Many modern machine learning software frameworks represent neural networks as a dataflow graph. In a dataflow graph, the nodes represent units of computation, and the edges represent the data consumed or produced by a computation. Representing a CNN in this form allows the underlying software framework to optimize the execution of math operations through increased parallelism, distributed execution, and compiler-generated optimizations. One way of creating a persistent representation of a neural network is to save the dataflow graph in a standardized machine-readable form. Some of the nodes in the neural network dataflow graph may have parameters θ_i that are optimized when training the CNN. These parameters are generally referred to collectively as model weights. It follows that both the dataflow graph and the associated weights are required to fully represent a trained neural network. An example of a dataflow graph for a single-layer neural network is shown in Figure 1.

In this paper, we propose a standardized representation of CNN based on the dataflow graph model and the existing PMML standard. In our proposed format, each layer is represented as an interconnected node in the dataflow graph. Model weights are

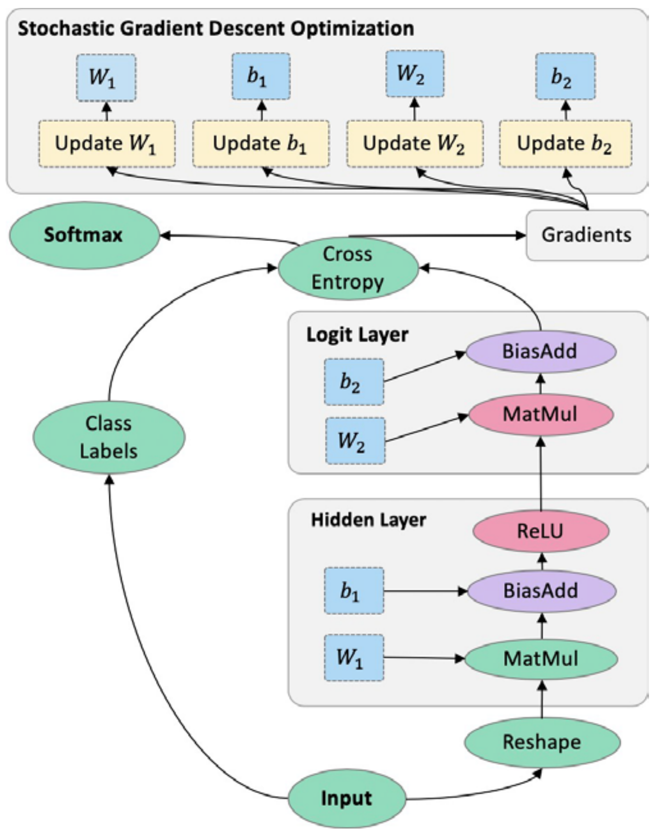

FIGURE 1. DATAFLOW GRAPH FOR TRAINING A SIMPLE NEURAL NETWORK. VARIABLES W_1 AND W_2 ARE THE NEURAL NETWORK WEIGHTS. VARIABLES b_1 AND b_2 ARE THE BIASES.

stored in a separate file, and associated with each node through a unique name. The primary contribution of the paper is the proposed PMML representation for CNN. Additional contributions include a study of the performance characteristics of this proposed format in the context of manufacturing defect detection, and PMML scoring engine.

The remainder of the paper is organized as follows. We begin by exploring related work in the field of standardized models. We then describe our proposed representation, and discuss in detail the mathematical background of each proposed layer. Next, we leverage transfer learning to train ten different CNN models for the task of casting defect detection and represent each trained model in the proposed PMML format. Finally, we develop a high-performance PMML scoring engine for evaluating new input images against PMML models. We show how the scoring engine can be used to evaluate manufacturing images on different hardware devices. The paper is concluded with a brief discussion and conclusion.

RELATED WORKS

There is a large body of work that discusses the use of CNNs for quality control [1,4,8], automated manufacturing [9], and additive manufacturing processes [10]. However, the core focus of this paper is to explore standardized representations for modern CNNs, and demonstrate how such representations can be beneficial to the manufacturing industry. For the remainder of this section, we describe a number of ways that machine learning models are currently stored and distributed.

PMML: PMML provides an open standard for representing data-mining and predictive models [11]. Once a machine-learning model has been trained in an environment like MATLAB, Python, or R, it can be saved as a PMML file. The PMML file can then be moved to a production environment, such as an embedded system or a cloud server. The code in the production environment can parse the PMML file, and use it to generate predictions for new unseen data points. It is important to note that PMML does not control the way the model is trained. PMML is purely a standardized way to represent the trained model.

ONNX: The Open Neural Network Exchange (ONNX) format is a community project created by Facebook and Microsoft [12]. ONNX provides a definition of an extensible dataflow graph model, as well as definitions of built-in operators and standard data types. Each dataflow graph is structured as a list of nodes that form an acyclic graph. Nodes have one or more inputs and one or more outputs. Each node is a call to an operator. Operators are implemented externally to the graph, but the set of built-in operators are portable across frameworks. Every framework supporting ONNX will provide implementations of these operators on the applicable data types.

Keras: Keras is a high-level open source neural network framework written in Python. It is capable of running on top of TensorFlow or Microsoft Cognitive Toolkit. The Keras framework has support for saving and restoring dataflow graphs and model weights. The dataflow graph for any Keras model can be exported to the Javascript object notation (JSON) format. Model weights can be saved in the compact Hierarchical Data format (HDF5) binary format. While these two options provide an easy way to save and distribute Keras models, they do not facilitate interoperability between different tools.

TensorFlow: TensorFlow is an open source software library for high-performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms. Like Keras, TensorFlow also provides an option for saving dataflow graphs and model weights. The dataflow graph can be exported to a binary file using the Protocol Buffer format. The model weights can be exported to a binary format using a binary format, likely based on the Protocol Buffer format.

PyTorch: PyTorch is an open source machine learning library for Python, based on Torch, used for applications such as natural language processing. It is primarily developed by Facebook's artificial-intelligence research group. PyTorch models are exported to a binary format using the Python Pickle protocol. While this approach provides a seamless method of persisting PyTorch models, it greatly hinders interoperability. In particular, relying on the Pickle object serialization format makes it difficult to transfer PyTorch models to other computing environments that do not support the Python runtime.

Given the current industry practices in machine learning, it appears that there is a need for greater standardization in the representation of deep neural networks. The creation and adoption of such standardized approaches is necessary to promote interoperability of machine learning frameworks and related tools.

PMML FOR CONVOLUTIONAL NEURAL NETWORKS

In this section, we describe how the existing PMML standard is extended to represent CNNs. Figure 2 shows the general structure of a PMML document, which includes four basic elements, namely, header, data dictionary, data transformation, and the data-mining or predictive model [5]. The *Header* element provides a general description of the PMML document, including name, version, timestamp, copyright, and other relevant information for the model-development environment. The *DataDictionary* element contains one or more *DataField* child elements which describe the data fields and their types, as well as the admissible values for the input data. We propose that *DataField* elements with a *dataType* attribute set to “image” can be used to represent images, however further work is needed to formalize this change. In addition, in a classification model, all of the class names are stored in the *DataDictionary* element.

Data transformation is performed using the optional *TransformationDictionary* or *LocalTransformations* element. These elements describe the mapping of the data, if necessary, into a form usable by the mining or predictive model. While support for image transformations would be highly beneficial to the proposed standard, it is outside the scope of this paper, and is left as a topic for future work. The last element in the general structure contains the definition and description of the predictive model. The element is chosen among a list of models defined in PMML standard. We propose the *DeepNetwork* model element as a new element for representing deep neural network models in PMML.

DeepNetwork Element: A CNN model is represented by a *DeepNetwork* element defined in the XML schema, which contains all the necessary information to fully characterize the model. The *DeepNetwork* element has a number of optional attributes that can be used to provide additional metadata about the model, such as the optimization algorithm used to optimize the hyperparameters. Figure 2 shows the elements which can be nested within the *DeepNetwork* element. The *DeepNetwork* element must contain one or more *NetworkLayer* elements which describe individual nodes in the dataflow graph. We now describe a set of layers which are required to minimally represent most of the common CNN architectures that have been proposed to date.

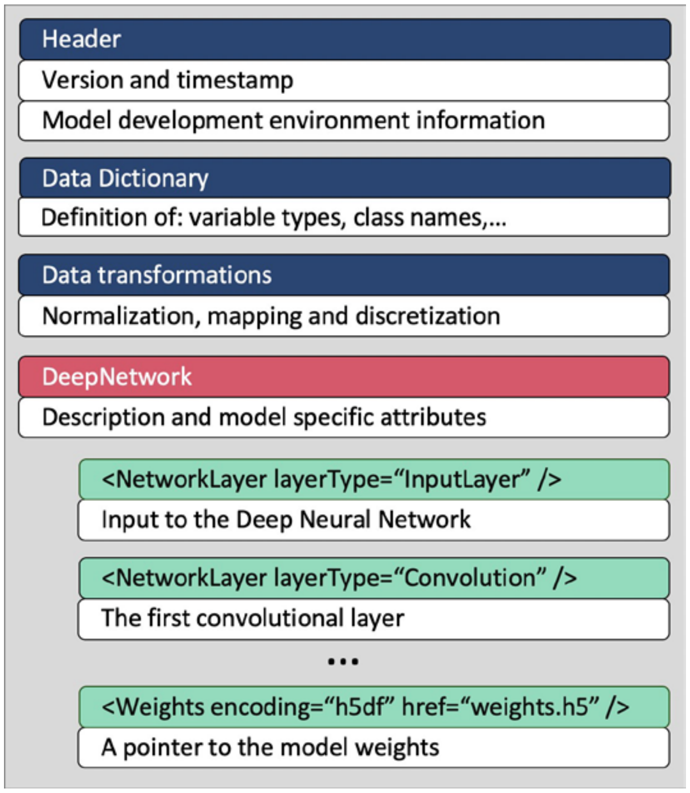
Header
Version and timestamp
Model development environment information
Data Dictionary
Definition of: variable types, class names,...
Data transformations
Normalization, mapping and discretization
DeepNetwork
Description and model specific attributes
<NetworkLayer layerType="InputLayer" />
Input to the Deep Neural Network
<NetworkLayer layerType="Convolution" />
The first convolutional layer
• • •
<Weights encoding="h5df" href="weights.h5" />
A pointer to the model weights

FIGURE 2. THE STRUCTURE AND CONTENTS OF A DEEP-NETWORK PMML FILE.

Convolution Layer: The convolution layer convolves a convolutional kernel with the input tensor. The cardinality of the convolution tensor must be equal to that of the input tensor. The size of the convolutional kernel is governed by the *KernelSize* child element, and the stride is governed by the *KernelStride* child element. An activation function can be optionally applied to the output of this layer. An example of a convolution layer in the proposed PMML format is shown in Figure 3.

Merge Layer: The merge layer takes two tensors of equal dimensions and combines them using an elementwise operator. Allowable operator types are “addition”, “subtraction”, “multiplication”, “division”. An example of a merge layer in the proposed PMML format is shown in Figure 4.

Dense Layer: The dense layer represents a fully connected neural network layer. An activation function can optionally be applied to the output of this layer. An example of a dense layer in the proposed PMML format is shown in Figure 5.

Concatenation Layer: The concatenation layer takes two tensors and concatenates them along a given dimension. The cardinality of the two tensors must be equal. The size of all dimensions other than the concatenation dimension must be equal.

```
<NetworkLayer
activation="relu"
layerType="Conv2D"
name="block1_conv1"
padding="same"
use_bias="True">
     <InboundNodes>
         <Array n="1" type="string">input_1</Array>
     </InboundNodes>
     <ConvolutionalKernel channels="64">
        <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>
```

FIGURE 3. CONVOLUTION LAYER EXAMPLE.

```
<NetworkLayer
layerType="Merge"
axis="3"
operator="add"
name="conv2_block1_concat">
  <InboundNodes>
    <Array n="2" type="string">pool1 conv2</Array>
  </InboundNodes>
</NetworkLayer>
```

FIGURE 4. MERGE LAYER EXAMPLE.

```
<NetworkLayer
activation="softmax"
channels="1000"
layerType="Dense"
name="fc1000">
  <InboundNodes>
    <Array n="2" type="string">pool1 conv2</Array>
  </InboundNodes>
</NetworkLayer>
```

FIGURE 5. DENSE LAYER EXAMPLE.

An example of a concatenation layer in the proposed PMML format is shown in Figure 6.

Pooling Layer: The pooling layer applies a pooling operation over a single tensor. The width of the pooling kernel is governed by the *PoolSize* child element, and the stride is governed by the *Strides* child element. The pooling operation can be either "max" or "average" depending on the value of the *operation* attribute.

```
<NetworkLayer
layerType="Concatenate"
axis="3"
name="conv2_block1_concat">
  <InboundNodes>
    <Array n="2" type="string">pool1 conv2</Array>
  </InboundNodes>
</NetworkLayer>
```

FIGURE 6. CONCATENATION LAYER EXAMPLE.

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 or input channels, allowing values across multiple channels to be combined to form the output. Depthwise convolutions keep each channel separate - hence the name depthwise.

Batch Normalization Layer: The batch normalization layer applies a batch normalization operation to the input tensor, which aims to generate an output tensor with a zero mean and unit variance. The linear transformation between input and output is based on the distribution of inputs at test time, and the parameters are generally fixed after training is completed.

Activation Layer: The activation layer applies an activation function to each element in the input tensor. The activation function can be any one of "relu", "sigmoid", "tanh", "selu", "elu", "softmax". The *threshold* attribute allows the activation function to be offset horizontally. The *max_value* attribute limits the maximum value of each output value.

Global Pooling Layer: This layer applies a pooling operation across all spatial dimensions of the input tensor. The pooling operation can be either "max" or "average", and is specified using the *operation* attribute. This layer returns a tensor that has size (*batch_size*, *channels*).

Zero Padding Layer: This layer pads the outside of a 2D tensor with zeros. This operation is commonly used to increase the size of oddly shaped layers, to allow dimension reduction in subsequent layers. The number of zeros that are added to each dimension is specified using the *padding* attribute.

Reshape Layer: The 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.

Flatten Layer: The flatten layer flattens the input tensor such that the output size is (*batch_size*, *n*) where *n* is the number of values in the input tensor.

Many of these layers have weights that are optimized during the training process. These weights are stored in a binary key-value format, such as HDF5, and the corresponding file is referenced using the *Weights* element. The *Weights* element has *encoding*, *checksum* and *href* attributes which specify the encoding, checksum and location of the model weights, respectively.

Examples of full PMML models for many common CNN architectures are made available at https://github.com/maxkferg/python-pmml [13].

PMML FOR DEFECT DETECTION

In this section, we show how the proposed PMML format can be used to facilitate transfer learning in a defect detection task. CNNs have been particularly successful for detecting manufacturing defects using camera or Xray images [1, 2, 14]. Historically, manufacturing researchers and technicians were concerned that deep learning systems would require an infeasible amount of data to train. However, by leveraging transfer learning, it is now possible to train a powerful computer vision system with as little as a thousand training examples. In transfer learning, a neural network model is first trained on a large dataset, such as the ImageNet dataset. The trained model is then fine-tuned on a smaller domain-specific dataset, such as a casting defect dataset. Hence, the successful application of transfer learning is highly dependent on the availability of high-quality neural network models. Without a standardized representation of neural network models, it is challenging for researchers and practitioners to share their pretrained models.

A total of ten pretrained image classification models are converted to the PMML format, allowing them to be easily loaded into machine learning frameworks such as Keras, TensorFlow or PyTorch. Each model is then fine-tuned on the GDXray casting defect dataset. The fine-tuned models are exported to the proposed PMML format and made publicly available. Finally, the PMML models are loaded onto a TensorFlow scoring engine to demonstrate how the model could be used in a manufacturing environment.

GDXray dataset: The GDXRay dataset is a collection of annotated Xray images [15]. The Castings series of this dataset contains 2727 X-ray images mainly from automotive parts, including aluminum wheels and knuckles. The casting defects in each image are labeled with tight fitting bounding-boxes. The size of the images in the dataset ranges from 256×256 pixels to 768×572 pixels. The GDXray dataset has been used by many researchers as a standard benchmark for defect detection including [16], where patches of size 32×32 pixels are cropped from the GDXray Castings series and used to test a number of different classifiers. The best performance is achieved by a simple local binary pattern (LBP) descriptor with a linear support vector machine (SVM) classifier [16]. Several deep learning approaches are also evaluated, obtaining up to 86.4 % patch classification accuracy. We train multiple different CNN models to classify

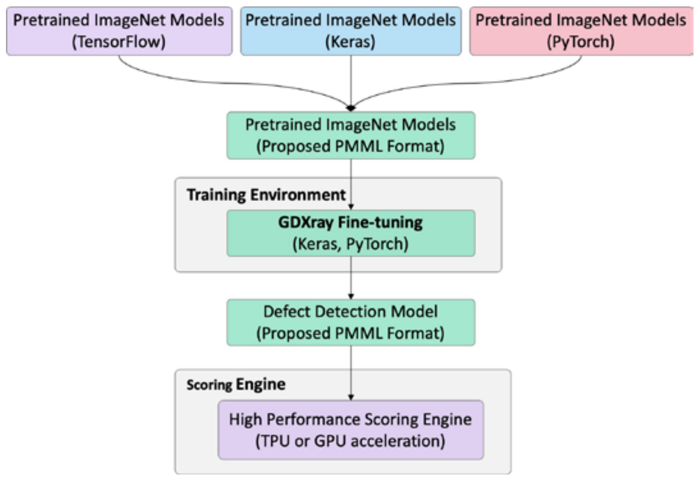
Pretrained ImageNet Models (TensorFlow)
Pretrained ImageNet Models (Keras)
Pretrained ImageNet Models (PyTorch)
Pretrained ImageNet Models (Proposed PMML Format)
Training Environment
GDXray Fine-tuning (Keras, PyTorch)
Defect Detection Model (Proposed PMML Format)
Scoring Engine
High Performance Scoring Engine (TPU or GPU acceleration)

FIGURE 7. TRANSFER LEARNING WITH THE PROPOSED PMML FORMAT. THE PRETRAINED MODELS ARE FIRST CONVERTED TO THE PROPOSED PMML FORMAT AND FINE-TUNED ON THE GDXRAY DEFECT TILE DATASET. THE MODELS ARE THEN TRANSFERRED TO A SCORING ENGINE USING THE PROPOSED FORMAT.

image tiles from the GDXray casting dataset. Each image in the casting set is divided into 224×224 pixel tiles. When an image cannot be divided perfectly into 224×224 pixel tiles, it is first padded with black pixels and then divided into tiles. Image tiles containing one or more casting defects are considered defective ($y = 1$), otherwise the tile is considered satisfactory and assigned class $y = 0$. The goal is to develop a classifier that can correctly predict the class of an unseen tile. To ensure the results are consistent with previous work, the training and testing data is divided in the same way as described in [2]. The preprocessed training dataset contains 8192 tiles. A further 1000 tiles from the training set are assigned to the dev set, which is used to test whether the model is overfitting. The test set contains a total of 1894 tiles.

Machine learning models: Ten different CNN models are trained on the GDXray tiles. Four different CNN architectures are trained, namely the VGG (Visual Geometry Group), Residual Network (ResNet), MobileNet and DenseNet architectures. Pretrained ImageNet models are converted to PMML from the native Keras and PyTorch model formats. The framework-agnostic nature of PMML provides the flexibility to train and save models in any supported language and machine learning framework. The multi-step training process is illustrated in Figure 7. The CNN models are trained on the GDXray dataset using the Keras machine learning framework. Additionally, the VGG and ResNet models are also trained on the GDXray dataset using the PyTorch machine learning framework.

The sparse occurrence of manufacturing defects often creates a challenge when training machine learning models. In

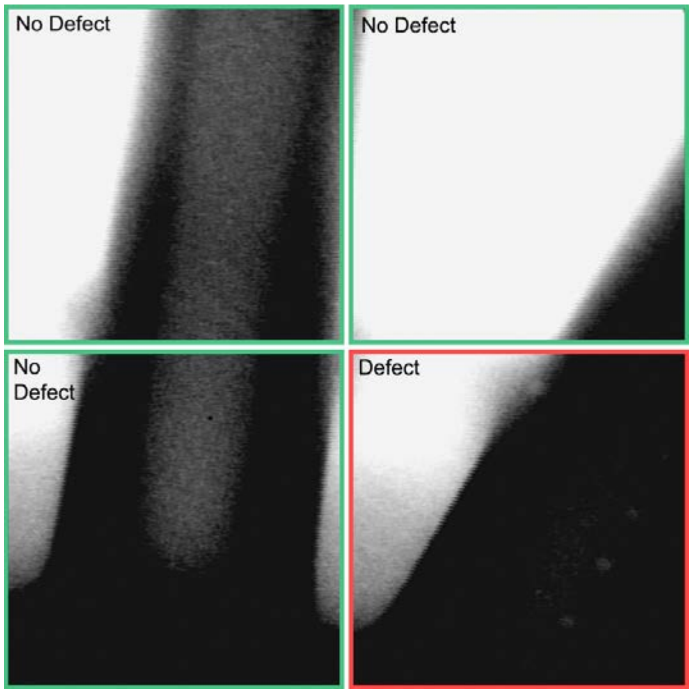

FIGURE 8. PREDICTION RESULTS FOR FOUR TILES FROM THE GDXRAY DEFECT TILE DATASET. EACH 224 $\times$ 224 PIXEL TILE WAS CLASSIFIED INDIVIDUALLY, AND THEN THE TILES WERE RECOMBINED TO FORM THE FIGURE.

highly imbalanced datasets, predictive models can often minimize loss by simply predicting the most common class. To avoid this pitfall, defective images are oversampled from the dataset. Specifically, we ensure that at least one defective tile is sampled from the dataset for every three clean tiles. Data augmentation is applied to reduce potential overfitting. During training, each image is rotated 90 degrees with probability 0.5, flipped horizontally with probability 0.5 and flipped vertically with probability 0.5. A small amount of Gaussian random noise is also added to each image. The gradient-based Adam algorithm is used for parameter optimization. The dense layers of each model are trained on the GDXray dataset for 10 epochs with a learning rate of 0.005, while keeping the weights of the convolutional layers fixed. The model is then trained for an additional 10 epochs without fixing any of the weights, using a learning rate of 0.001. After each epoch, the model is saved to PMML and tested on the dev set. The model that achieves the highest performance on the dev set is selected as the final model. The test set prediction accuracy for each model is presented in Table 1. Some example predictions are shown in Figure 8. The trained ImageNet classification models and GDXray defect classification models are made publicly available in the proposed PMML format, to accelerate future research in this direction [13].

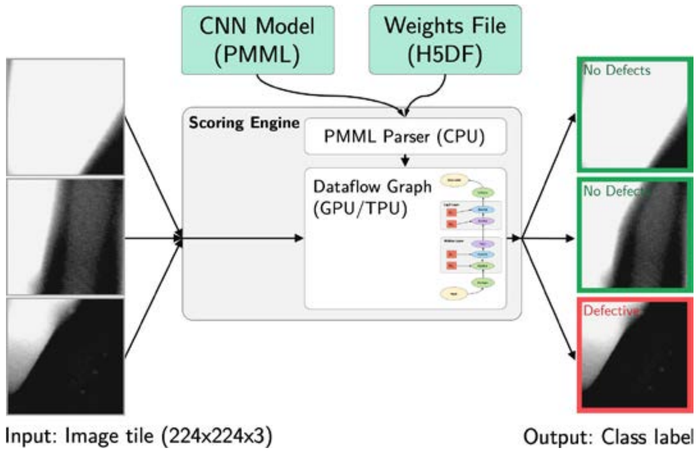

FIGURE 9. NEURAL NETWORK EVALUATION WITH THE PROPOSED PMML FORMAT. A HIGH-PERFORMANCE SCORING ENGINE EVALUATES NEW IMAGES AGAINST THE MODEL AND RETURNS THE PREDICTED CLASS.

EFFICIENCY AND PERFORMANCE

Modern CNNs are growing increasingly complex, with recent architectures having hundreds of layers and millions of parameters. Therefore, storage efficiency, serialization performance and deserialization performance must be considered when designing a standardized format for modern neural networks. To evaluate the performance of the proposed format, we develop a PMML scoring engine with support for deep CNN models, and conduct a number of performance experiments.

There are two main factors when considering the performance of a scoring engine: (1) The amount of time it takes to load a model from a PMML file into memory, and (2) the amount of time required to evaluate a new input. We will refer to (1) as the initial load time and (2) as the evaluation time. The initial load time of modern neural networks tends to be quite large as most modern CNNs have complicated dataflow graphs and millions of parameters. However, models are generally kept in memory between subsequent predictions, so initial load time is not a major influence on performance. The prediction time, however, is critical to many applications in manufacturing, as it dictates the amount of time between an observation being made, and a prediction being obtained.

A scoring engine is developed to evaluate image inputs against CNN models in the proposed PMML format. The scoring engine uses the TensorFlow machine learning framework to perform the mathematical operations associated with each network layer. In our scoring engine, the PMML file is parsed using the lxml XML parser [17] and the Python programming language. The scoring engine is engineered in a way such that the dataflow graph can be evaluated on a central processing unit (CPU), graphics processing unit (GPU), or tensor processing unit

TABLE 1. TEST ACCURACY AND MODEL STATISTICS FOR TEN CNN MODELS TRAINED ON THE GDXRAY DEFECT TILE DATASET.

Model	Training Framework	PMML Size (KB)	Weights Size (MB)	Parameters	Training time (s)	Test accuracy
VGG-16	PyTorch	58	528	138,357,544	1442	0.899
VGG-19	PyTorch	60	549	143,667,240	2301	0.921
ResNet-50	PyTorch	106	99	25,636,712	1349	0.933
ResNet-152	PyTorch	154	244	60,344,232	2012	**0.944**
VGG-16	Keras	54	528	138,357,544	1322	0.899
VGG-19	Keras	56	549	143,667,240	1943	0.921
ResNet-50	Keras	101	99	25,636,712	1201	0.933
ResNet-152	Keras	156	244	60,344,232	1894	**0.940**
MobileNet-224	Keras	75	16	4,253,864	1561	0.895
DenseNet-121	Keras	185	33	8,062,504	2104	0.942

(TPU). The scoring engine can be used to evaluate batches of one or more images against a PMML model as shown in Figure 9.

The performance characteristics of the proposed PMML format are analyzed using the aforementioned PMML scoring engine. Experiments are conducted using four common CNN architectures, namely VGG-16, ResNet-152, DenseNet-121 and MobileNet-224. In each case, we use a PMML model trained on the GDXray defect detection task. The experiments are conducted on three platforms: A desktop computer with a single NVIDIA 1080 Ti GPU, an identical desktop computer without a dedicated GPU, and a cloud-based virtual machine with a tensor processing unit (TPU). Figure 10 shows the total amount of time required to build the dataflow graph and load the model weights from file. Figure 11 shows the time required to parse the PMML file and load the dataflow graph into memory, on each of the three platforms. Figure 12 shows the time required to load the neural network weights into memory. Finally, Figure 13 shows the prediction time on the three platforms.

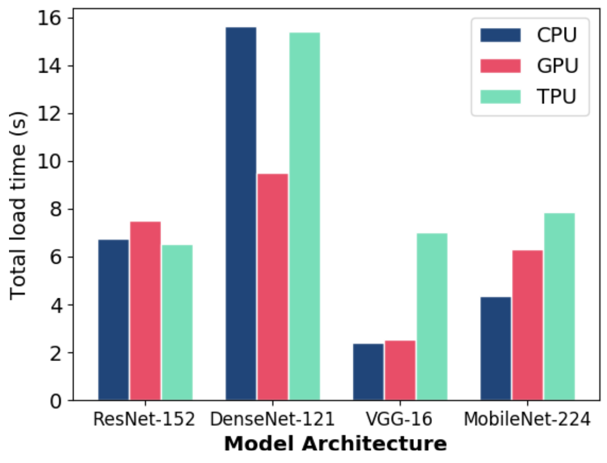

FIGURE 10. TOTAL LOAD TIME FOR FOUR DIFFERENT MODELS. TOTAL LOAD TIME IS DEFINED AS THE TIME FROM WHEN THE PMML FILE FIRST STARTS TO LOAD UNTIL THE TIME THAT THE DATAFLOW GRAPH IS READY TO MAKE PREDICTIONS.

DISCUSSION

In this work, we proposed an extension to the PMML format for the standardized representation of CNN models. The proposed extension adds a DeepNetwork element to the existing standard. The proposed format describes the neural network architecture using the human-readable XML format, making it practical to inspect the architecture of trained models. The human-readable nature of this format is highly beneficial when sharing models trained by researchers, competition teams or large industry teams; rather than documenting the model details in a white-paper, the PMML model can be used to express the architecture in a human-readable manner.

An alternative approach for storing CNN models is to store the entire dataflow graph in a binary format. This approach is currently being employed as part of the ONNX standard. The main benefit of storing the entire dataflow graph is that it pro-

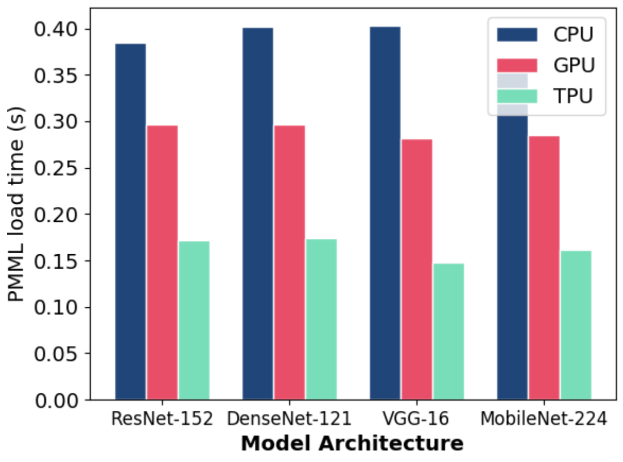

FIGURE 11. PMML LOAD TIME FOR FOUR DIFFERENT MODELS. PMML LOAD TIME IS DEFINED AS THE TIME TAKEN TO LOAD AND PARSE THE PMML FILE.

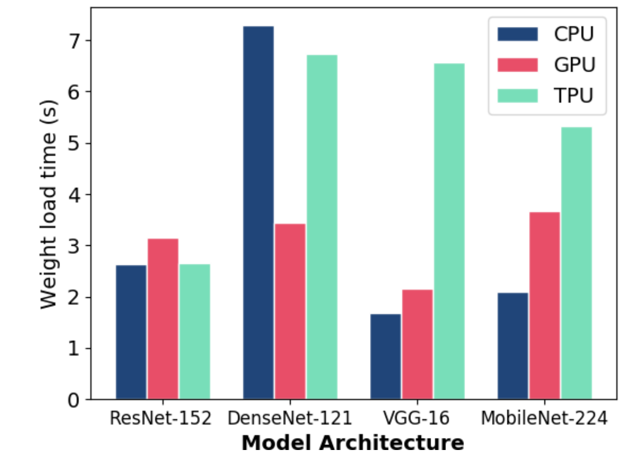

FIGURE 12. WEIGHT LOAD TIME FOR FOUR DIFFERENT MODELS. WEIGHT LOAD TIME IS DEFINED AS THE TIME TAKEN TO BUILD THE DATAFLOW GRAPH ON THE EXECUTION DEVICE AND LOAD THE MODEL WEIGHTS.

vides more flexibility for custom mathematical operations. However, binary formats such as ONNX are not human-readable, making them more difficult to interpret. Both methods could be useful in different contexts: The proposed PMML representation could be particularly useful when widely-used neural network architectures, such as ResNet, are trained on a novel task and shared across the research or industry communities. A binary format such as ONNX is likely more useful for representing complex neural network architectures such as recurrent neural networks.

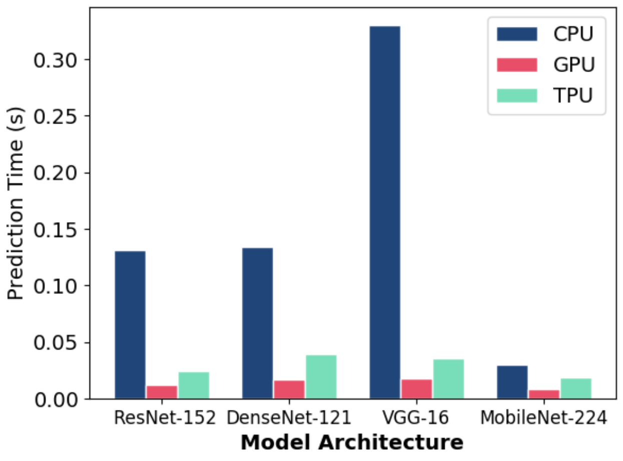

FIGURE 13. MODEL PREDICTION TIME ON THREE DIFFERENT COMPUTATIONAL PLATFORMS. MODEL PREDICTION TIME IS DEFINED AS THE AMOUNT OF TIME REQUIRED TO GENERATE A PREDICTION FROM A NEW INPUT, WHEN USING A BATCH SIZE OF 1.

The neural network weights are critical to fully representing a trained neural network; without these weights a prediction cannot be calculated. While PMML already contains support for a *NeuralNetwork* model element, this requires every neural network node to be specified. Specifying a 224×224 convolution layer with a 3×3 kernel requires 9 weights using the *DeepNetwork* element compared to $224 \times 224 \times 3 \times 3 = 451,584$ weights with the traditional *NeuralNetwork* element. In this work, we chose to use the HDF5 format to store model weights, as it provides a simple and efficient method of storing a map between layer names and weight tensors. However, model weights could also be saved using a binary format like ONNX, or directly embedded in the PMML file as a base64 string.

A tile-base casting defect classifier was developed to illustrate how the proposed PMML format makes transfer learning more accessible to a large number of machine learning frameworks. The ResNet-152 model obtained 94.4 % prediction accuracy on the test set, outperforming the highest scoring CNN model in [16], ImageXnet, which achieved 86.4 % accuracy. The improved performance is likely due to the use of larger tiles (224×224 pixels) compared to the 32×32 pixel patches used in ImageXnet.

Scoring engines will become increasingly important in the smart manufacturing industry, especially as internet-connected manufacturing machines become more mainstream. For many real-time applications, the response time of the scoring engine must be sufficiently low to avoid manufacturing devices becoming idle whilst waiting for feedback from the scoring engine. Due to the declarative nature of the proposed PMML format, it is possible for our scoring engine to evaluate models on CPU, GPU or TPU hardware. This level of flexibility could be highly beneficial for manufacturing applications, where models must be evaluated on a range of hardware systems.

CONCLUSION

In this work, we proposed an extension to the PMML format for a standardized representation of convolutional neural network models. A tile-base casting defect classifier was developed to illustrate how the proposed PMML format makes transfer learning more practical on many machine learning frameworks. A scoring engine was created for this new standardized schema and used to evaluate the performance characteristics of the proposed format. The scoring engine and the trained models have all been made publicly available to accelerate research in the field.

ACKNOWLEDGMENT

This research is partially supported by the Smart Manufacturing Systems Design and Analysis Program at the National Institute of Standards and Technology (NIST), Grant Number 70NANB18H193 awarded to Stanford University.

DISCLAIMER

Certain commercial systems are identified in this paper. Such identification does not imply recommendation or endorsement by NIST; nor does it imply that the products identified are necessarily the best available for the purpose. Further, any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NIST or any other supporting U.S. government or corporate organizations.

REFERENCES

[1] Ferguson, M., Ak, R., Lee, Y.-T. T., and Law, K. H., 2018. “Detection and segmentation of manufacturing defects with convolutional neural networks and transfer learning”. *Smart and Sustainable Manufacturing Systems (SSMS)*, ***2***(1), Oct.

[2] Ferguson, M., Ak, R., Lee, Y.-T. T., and Law, K. H., 2017. “Automatic localization of casting defects with convolutional neural networks”. *IEEE International Conference on Big Data (Big Data 2017)*, IEEE, pp. 1726–1735.

[3] Napoletano, P., Piccoli, F., and Schettini, R., 2018. “Anomaly detection in nanofibrous materials by CNN-based self-similarity”. *Sensors*, ***18***(1), p. 209.

[4] Chu, Y., Huang, C., Xie, X., Tan, B., Kamal, S., and Xiong, X., 2018. “Multilayer hybrid deep-learning method for waste classification and recycling”. *Computational Intelligence and Neuroscience*, ***2018***.

[5] Guazzelli, A., Zeller, M., Lin, W.-C., Williams, G., et al., 2009. “PMML: An open standard for sharing models”. *The R Journal*, ***1***(1), pp. 60–65.

[6] Gorea, D., 2008. “Dynamically integrating knowledge in applications. An online scoring engine architecture”. *Advances in Electrical and Computer Engineering*, ***8***(15), pp. 44–49.

[7] Hecht-Nielsen, R., 1992. “Theory of the backpropagation neural network”. *Neural Networks for Perception*. Elsevier, pp. 65–93.

[8] Hanzaei, S. H., Afshar, A., and Barazandeh, F., 2017. “Automatic detection and classification of the ceramic tiles surface defects”. *Pattern Recognition*, ***66***, pp. 174 – 189.

[9] Allard, U. C., Nougarou, F., Fall, C. L., Giguère, P., Gosselin, C., Laviolette, F., and Gosselin, B., 2016. “A convolutional neural network for robotic arm guidance using semg based frequency-features”. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 2464–2470.

[10] Shevchik, S. A., Kenel, C., Leinenbach, C., and Wasmer, K., 2018. “Acoustic emission for in situ quality monitoring in additive manufacturing using spectral convolutional neural networks”. *Additive Manufacturing*, ***21***, pp. 598–604.

[11] Zeller, M., Grossman, R., Lingenfelder, C., Berthold, M. R., Marcade, E., Pechter, R., Hoskins, M., Thompson, W., and Holada, R., 2009. “Open Standards and Cloud Computing: KDD-2009 Panel Report”. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, pp. 11–18.

[12] Candela, J. Q., 2017. Facebook and Microsoft introduce new open ecosystem for interchangeable AI frameworks. Available at https://fb.me/candela_2017. Accessed May 22, 2019.

[13] Ferguson, M., 2019. Python PMML scoring engine and CNN models. Available at https://github.com/maxkferg/python-pmml/. Accessed February 19, 2019.

[14] Ren, R., Hung, T., and Tan, K. C., 2018. “A generic deep-learning-based approach for automated surface inspection”. *IEEE Transactions on Cybernetics*, ***48***(3), pp. 929–940.

[15] Mery, D., Riffo, V., Zscherpel, U., Mondragn, G., Lillo, I., Zuccar, I., Lobel, H., and Carrasco, M., 2015. “GDXray: The database of Xray images for nondestructive testing”. *Journal of Nondestructive Evaluation*, ***34***(4), Nov., p. 42.

[16] Mery, D., and Arteta, C., 2017. “Automatic defect recognition in Xray testing using computer vision”. *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, IEEE, pp. 1026–1035.

[17] Behnel, S., Faassen, M., and Bicking, I., 2005. lxml: XML and HTML with Python.