AN12892

Transfer Learning and the Importance of Datasets

Rev. 0 — 06/2020

Application Note

Contents

1 Introduction

This application note focuses on transferring learning and datasets. The theoretical part is general and device agnostic. The use case chapters of this document work with a handwritten digit recognition application for the i.MX RT1060, explained in *Handwritten Digit Recognition Using TensorFlow Lite on RT1060* (document AN12603).

The original handwritten digit recognition model was trained on the official MNIST dataset. AN12603 describes how the dataset the original model was trained on was collected in a different manner than how the input numbers are received by the RT1060 application. Because of these differences, the accuracy of the model on the input numbers was much lower than on the MNIST numbers. However, collecting a better dataset and using transfer learning can solve this accuracy issue.

This document describes:

- What transfer learning is.
- Why datasets are important.
- How to gather a dataset.
- How to perform transfer learning in TensorFlow.
- A use case example, which aims to improve the performance of the application from AN12603.

2 Transfer learning

Transfer learning gives machine learning models the ability to apply past experience to quickly and more accurately learn to solve new problems. The process is similar to how a person who can roller skate would find it much easier to learn to ice skate than a person who had never tried any form of skating whatsoever. This technique has become very important in deep learning and as Andrew Ng, one of the most prominent figures in the deep learning community, predicted at the NIPS 2016 conference (Ruder, 2016), it will be the next driver of ML success (Ng, 2016). Among the typical commercial use cases are Natural Language Processing (NLP) and computer vision.

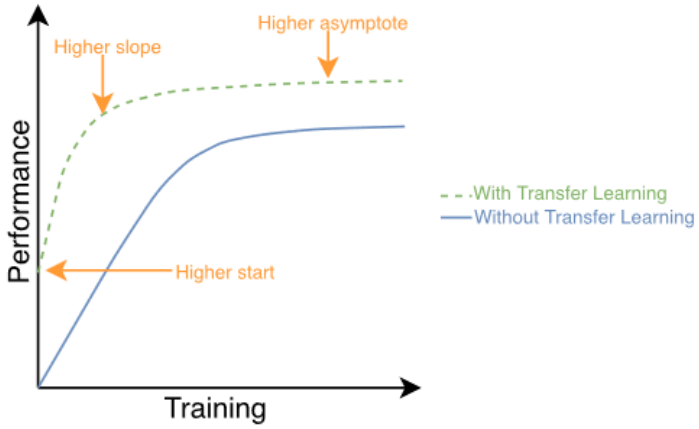

Figure 1. Training difference with transfer learning

The main advantages of transfer learning are improved accuracy even with small datasets and faster training due to a better starting point, as shown in Figure 1. An easy to explain example is image recognition. There is a dataset called ImageNet, which consists of over 15 million labeled images sorted into about 22,000 categories (Krizhevsky, Sutskever, & Hinton, 2012). It is used as a benchmark for an annual competition, which has given rise to some of the most popular convolutional neural networks today, like AlexNet, ResNet, Inception or GoogleNet. In these very deep models, the first layers look for very simple things, like diagonal, vertical and horizontal lines – similar to how classical image processing works, except that instead of humans designing the filters, the neural network learns them on its own. However, as the layers get deeper, they start to look for more and more complex shapes and in the final layers, the individual neurons often actually look for shapes even recognizable by the human eye, like beaks, archways, eyes or ears as can be seen in Figure 2. Due to the immense dataset, these neuron's filters are very precise and detailed.

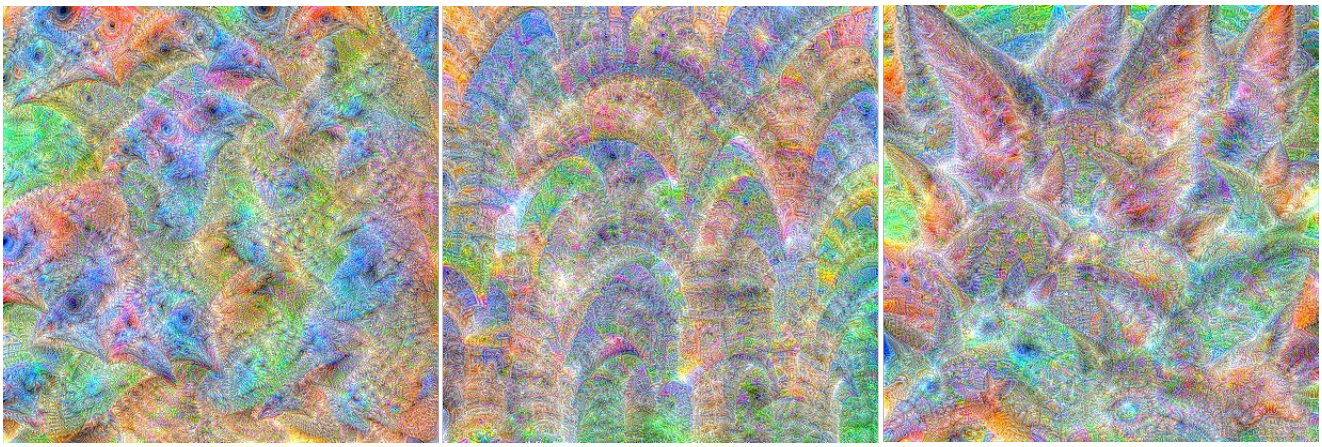

Figure 2. Convolutional Feature Visualization (Graetz, 2019)

Unfortunately, gathering and labeling a dataset as large as ImageNet and designing a complicated neural network is very time-consuming, difficult, and expensive. This is where transfer learning comes to the rescue. Instead of gathering and labeling millions of images and then creating a neural network from scratch, one can use the knowledge of existing pre-trained models and apply it to a new task. This is done by replacing the output layer or several of the last layers with one or more layers representing the new task, preventing all or some of the original layers from changing their weights and then training the whole model on the new dataset, which can be much smaller in size. By doing this, the new model can use the precise filters it already knows to look for features in the new dataset and thus it can converge to higher accuracies much faster than if it was initialized with random weights. Indeed, the already mentioned models like a pre-trained ResNet are very commonly used just for this purpose (Dongmei, Qigang, & Weiguo, 2018) (Ferreira et al., 2018) (Shaopeng, Guohui, & Yuan, 2019). Even so, transfer learning isn't a perfect solution to every problem. In the described case, the solution is more accurate if the two datasets are similar in nature. If they differ too much, the model might not provide accurate results.

There are several types of transfer learning for deep neural networks. Among these are:

- Domain Adaptation, where the source domains (or datasets) are different but related to each other – this is the example described in the previous paragraphs.
- Multitask Learning, during which several tasks are being solved simultaneously. This allows the model to improve based on the similarities and differences between the tasks.
- Zero-shot and one-shot learning, which require the model to solve a problem that it has never encountered or only encountered a few times before, respectively.

To learn more about transfer learning, you can try out the eIQ Transfer Learning Lab with i.MX RT (NXP Semiconductors, 2019) and i.MX 8 (NXP Semiconductors, 2019) from NXP. Additionally, there are many detailed third-party transfer learning guides. Among them is a comprehensive hands-on guide from towardsdatascience (Sarkar, 2018).

3 Importance of a high-quality dataset

Gathering a high quality dataset is the most important part of the entire machine learning process, especially in supervised learning. However, preparing a complex dataset is a lot of intense work. The first step is to decide what exactly the model must solve and what type of data can be gathered. A simplified end-to-end example goal could be to enable a car to recognize traffic signs from pictures taken by its own frontal camera. In order for the model to generalize well and work safely when deployed, it would need a dataset that properly represents its task in as many situations as possible. That includes different lighting and weather conditions, different angles from which the picture could be taken, etc. Careful attention must also be paid to any bias that could accidentally be introduced into the dataset. Otherwise the model could find patterns or features unrelated to the traffic signs themselves, wrongly assume they are important and base its decisions on the wrong information. For example, if all pictures of one type of sign were taken during winter and all pictures of another type of sign were taken during summer, the model could actually make its decisions based on whether there is snow in the background instead of looking at the sign itself (Ribeiro, Singh, & Guestrin, 2016).

Once the dataset is planned out, it has to be gathered, cleaned and labeled. The larger the dataset is, the better it can represent the problem and the better the model can learn to solve it. At the same time however, with increasing size come increasing expenses and time commitments. Several techniques were developed to deal with these issues. Among them are transfer learning, data augmentation, and data synthesis. Data augmentation and synthesis can artificially expand the dataset by adding simple or complex variations on already collected data or simulating new examples. That includes rotations, scaling, distortions, adding noise and different backgrounds etc. The cleaning part is to remove data, which either doesn't represent the problem or contains too much unrelated information that could confuse the model instead of helping it learn. The labeling is crucial to supervised learning. Data that is incorrectly labeled can degrade the accuracy of the model, especially if the mislabeling is done systematically (Fard, Hollensen, Mcilory, & Trappenberg, 2017). Both cleaning and labeling usually have to be done by humans and are difficult to automate.

When the dataset is ready, it must be split into training and testing parts. The testing part can then be further split into a testing set, which is used to tune the model and a validation set consisting of examples the model can never see during the training process. The validation set is then used to check how well the model works with new data. If the training set is too small, the model might overfit on it and lose the ability to generalize to data it never saw. If the training and testing sets were collected differently from the validation set, there might be a loss of accuracy due to data mismatch. Andrew Y Ng goes through the process of building a deep learning application in the already cited NIPS 2016 presentation (Ng, 2016).

There are services and open-source communities that focus on dataset gathering and labeling. Some include paid guidance and help customers with the whole process. Others provide free, community gathered and labeled datasets that anyone can use. A couple of examples are Quick, Draw! (Google, 2020) or DatasetList (Plesa, 2020). Lastly, like the already mentioned ImageNet or the many variations of MNIST, there are also datasets collected by researchers that are open to the public to help with further research and testing in deep learning and artificial intelligence in general.

4 Handwritten digit recognition use case

AN12603 focuses on creating a handwritten digit recognition application with a touch sensitive LCD mounted on an i.MX RT1060 EVK board. The convolutional neural network model used in the application was trained on the official MNIST dataset. However, even though the model achieved 99% accuracy on the MNIST dataset, it could only recognize the numbers written on the LCD

with a 72% success rate. This was due to the mismatch in how the input for training and regular operation was collected. The MNIST dataset contains edited photographs of digits written with a pen or pencil on a paper whereas the application takes as input a single pixel thick line drawn with a finger on an LCD screen. The application input is then further processed to resemble the MNIST pictures, but the results are never identical in nature.

Since digit recognition is a really simple task for neural networks, this problem could be effectively solved without transfer learning. It would be sufficient to gather a dataset with a few hundred or thousand examples for each digit, written by many different people, so that the model does not specialize on a specific person's handwriting. However, even in this case the results can be slightly improved by transfer learning from the original MNIST-trained model, especially when trained on only a few examples.

5 Gathering a custom MNIST-like dataset

For the purposes of gathering a new dataset, the GUI from the digit recognition application was slightly altered and reused for a new gathering application, as shown in Figure 3. This application generates a random number between 0 and 9. The user is required to write this number into four boxes and submit the inputs. These numbers are then processed the same way as the input in the digit recognition application and stored on an SD card. The SD card must be initialized beforehand to contain a folder for each digit named **0**, **1**, etc. The application stores each example in the expected folder based on the requested number. The examples are stored in a binary DAT format. The user must ensure the submitted examples are correct.

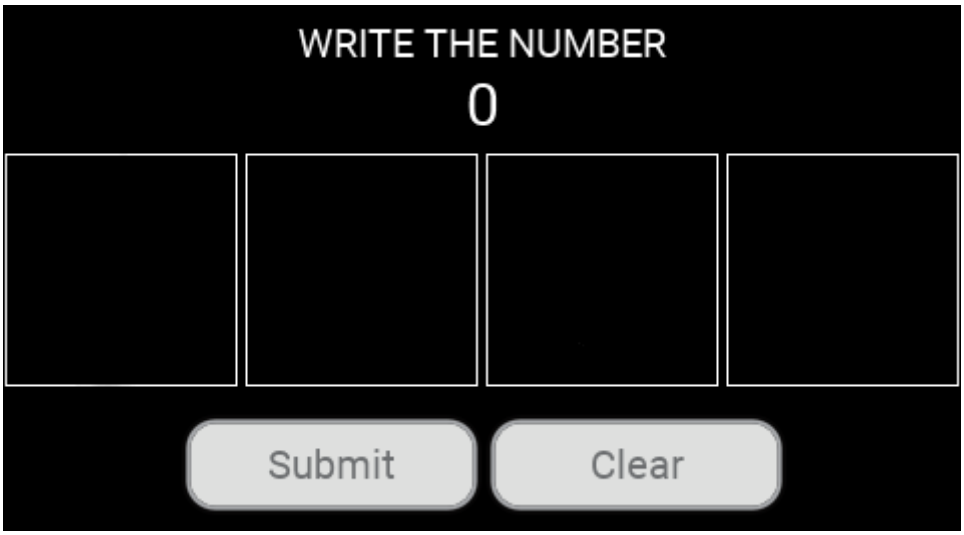

Figure 3. Dataset gathering application GUI

This application was left at the reception of a European NXP site for employees to use and contribute to the new custom MNIST dataset. The data collection was performed in two turns:

- First, a training and testing dataset was collected in one week, consisting of 4084 samples.
- The next week, a validation dataset was collected with 2388 samples.

Of the collected samples, several had to be removed or relabeled due to either being submitted to the wrong category, being too misshapen or not containing a digit at all, as shown in Figure 4.

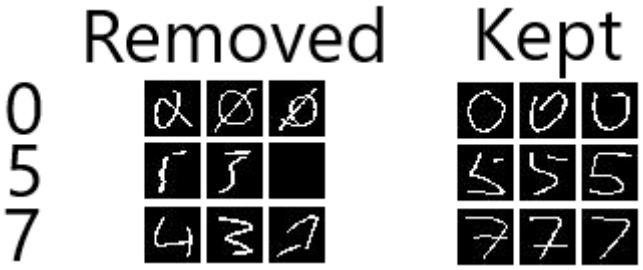

Figure 4. Examples of removed and kept submissions

Before cleaning, the data was converted from the DAT format, used for storing inputs in the application, to the JPG format. For this purpose, the `dat_to_jpg.py` script was created. This script automatically converts all the `.DAT` files to `.jpg` files and stores them in a new folder with a separate subfolder for each digit. The cleaning itself had to be performed manually because recognizing a correct example from an incorrect one automatically would require a separate AI application.

After cleaning, the original datasets were reduced to 3558 training and testing examples and 2212 validation examples. The training dataset contains around 350 examples for each digit but isn't perfectly balanced. The number **7** has the least amount of examples at 312, while the number **1** has the most at 437. The validation set is distributed similarly with around 220 examples per digit.

Balanced distribution of examples for each class is important for large datasets. In this case the imbalance didn't affect the transfer learning accuracy but it did have minor effects on models trained from scratch, as seen in Table 1. A script was created to automate the creation of balanced datasets from the original dataset. This script is called `get_x_jpgs.py` and it is able to extract a specified amount of random pictures from each class of the original dataset. If the specified amount exceeds the total amount of examples for one or more of the digits, the script fails and lets the user know this distribution can't be achieved. The resulting dataset is stored in a new folder with a separate subfolder for each digit.

6 Transfer learning in TensorFlow

1. Obtain a model to transfer knowledge from. For this application, the model from AN12603 Digital Lock software package is used. In order to use this model's knowledge, it is necessary to load its weights and biases into a new model of the same architecture.

   The script used for transferring learning is called `transfer_learning_mnist.py`. After importing the necessary libraries, the model architecture can be defined as shown in Figure 5.

```
image = tf.keras.layers.Input(shape=(28, 28, 1))

y = tf.keras.layers.Conv2D(filters=32,
                           kernel_size=5,
                           padding='same',
                           activation='relu')(image)
y = tf.keras.layers.MaxPooling2D(pool_size=(2, 2),
                                 strides=(2, 2),
                                 padding='same')(y)
y = tf.keras.layers.Conv2D(filters=32,
                           kernel_size=5,
                           padding='same',
                           activation='relu')(y)
y = tf.keras.layers.MaxPooling2D(pool_size=(2, 2),
                                 strides=(2, 2),
                                 padding='same')(y)
y = tf.keras.layers.Flatten()(y)
y = tf.keras.layers.Dense(1024, activation='relu')(y)
y = tf.keras.layers.Dropout(0.4)(y)

probs = tf.keras.layers.Dense(10, activation='softmax')(y)

model = tf.keras.models.Model(image, probs, name='mnist')
```

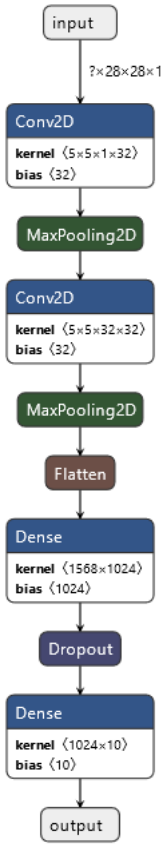

Figure 5. Model definition

2. Once defined, the original model's weights can be loaded in with the `load_model` function. In order to use all of the convolutional filters from the original model and to improve the decision making based on the new dataset, the convolutional layers must be frozen – to prevent their weights and bisases from changing, while the last three layers must be reset and allowed to learn. This can be done by removing the last three layers, freezing the whole model, and adding the last three layers back, as shown in Figure 6.

```
# Load pretrained weights
model = tf.keras.models.load_model(filepath=PRETRAINED_MODEL, compile=False)

# Remove the last 3 layers
model = tf.keras.models.Sequential(model.layers[:-3])

# Freeze the pretrained layers
for layer in model.layers:
  layer.trainable = False

# Add the last 3 layers uninitialized
new_layers = tf.keras.layers.Dense(1024, activation='relu')(model.layers[-1].output)
new_layers = tf.keras.layers.Dropout(0.4)(new_layers)
new_output = tf.keras.layers.Dense(10, activation='softmax')(new_layers)

# Put it all together
model = tf.keras.models.Model(model.layers[0].input, new_output, name='mnist_new')

# Compile
model.compile(
    optimizer=tf.keras.optimizers.Adam(LEARNING_RATE),
    loss='sparse_categorical_crossentropy',
    metrics=['sparse_categorical_accuracy'])
```

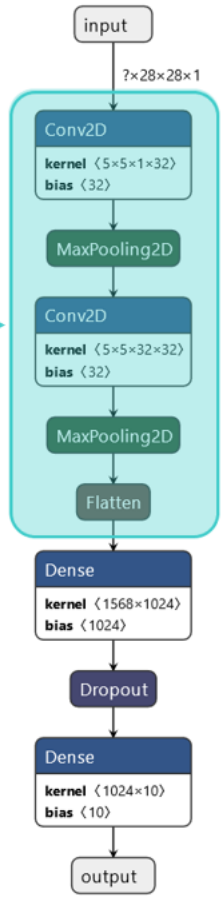

Figure 6. Transfer learning in TensorFlow with the built-in Keras library

3. To load the dataset, the `*.jpg` images are read from their corresponding subfolders as arrays, attached to a list of examples in the format of [image,class] and shuffled so that the examples are in a random order. Afterwards, the list is split into two, one for the images and the other for the classes. In this case, the classes can also be represented by labels. For example, the class for the number 0 has the label "0". Additionally, the images have to be normalized by dividing all of their pixel values by 255. Next, the list of images is reshaped into the input tensor expected by the model. This is performed through the numpy library and the function `reshape(-1, 28, 28, 1)`. The first number specifies the amount of inputs, **-1** means it can be as many as needed, the middle numbers specify the dimensions of the input, and the last number specifies the amount of channels the input has. The code is as shown in Figure 7.

```
training_data = []

for category in CATEGORIES:
   path = os.path.join(DATADIR, category)
   class_num = CATEGORIES.index(category)
   for img in os.listdir(path):
    img_array = cv2.imread(os.path.join(path, img), cv2.IMREAD_GRAYSCALE)
    training_data.append([img_array, class_num])

random.shuffle(training_data)

data = []
labels = []

for img, label in training_data:
 img = img.astype(np.float32)
 img = img/255.0
 data.append(img)
 labels.append(label)

data = np.array(data).reshape(-1, 28, 28, 1)
```

Figure 7. Loading the Dataset

4. Once everything is prepared, the model can be trained, as shown in Figure 8.

```
model.fit(
    data,
    labels,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE)
```

Figure 8. Training the model

7 Results

Several experimental runs were performed with different dataset distributions for both models trained with and without transfer learning. A statistic was compiled, as shown in Table 1. For the ten examples per digit experiment, the model was trained five times and validated on the whole validation set after each training. For the other distributions, the training was performed only three times because the variance between training accuracies wasn't as large. All training was done with a batch size of 32, except for the values in brackets, which uses a batch size of 16. Using the smaller batch size does not have any effect on the larger distributions.

Table 1. Validation accuracy table

Examples per digit	Validation set accuracy	
	With transfer learning	Without transfer learning
10	75.4-77.4% (78.3-80%)	71.6-72.1% (72.5-76.4%)
30	89.7-90%	85.3-86.3%
100	94-94.7%	91.2-92.8%

Table continues on the next page...

Table 1. Validation accuracy table (continued)

Examples per digit	Validation set accuracy	
	With transfer learning	Without transfer learning
300	96.8-97%	96-96.4%
ALL (~350)	96.8-97%	96.5-96.8%

Although using a relatively small dataset without transfer learning can yield large accuracy improvements over the original, the benefits of transfer learning can be seen even in this use case. As shown in Figure 9, the differences correspond accurately to the theoretical chart in Figure 1. Using transfer learning provides the model with a higher starting point and allows it to converge to high accuracies sooner.

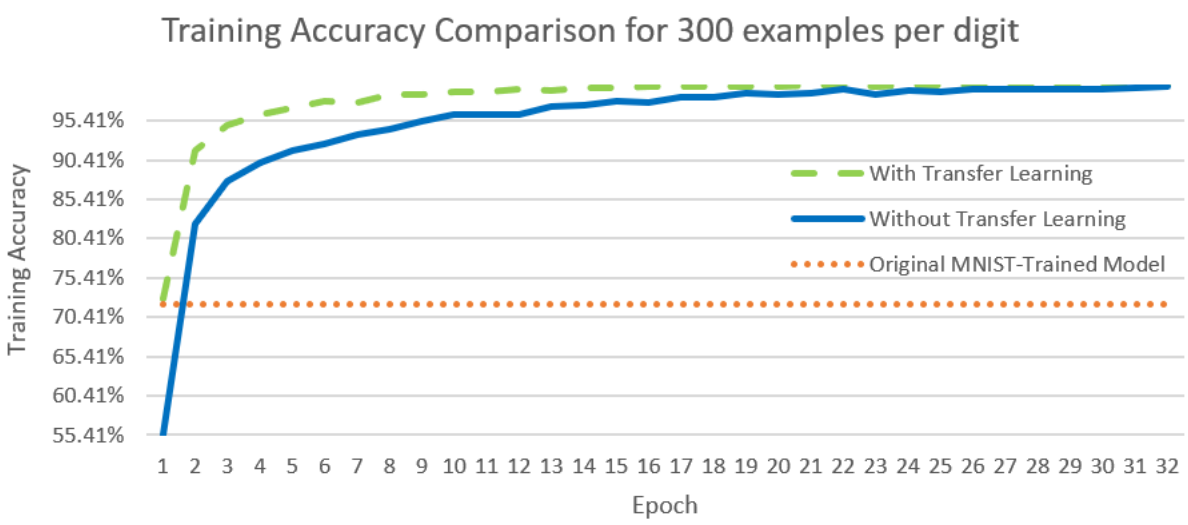

Figure 9. Training accuracy comparison graph

The `mnist_tflite_test.py` script was used to run the validation set tests. This script takes a model converted to TensorFlow Lite and a folder with the validation dataset `*.jpg` images, creates a dataset from the images the same way as the transfer learning script and runs inference on each of the pictures. It collects a statistic about how many images were predicted correctly, prints out which ones it predicted wrong and finally prints out the total accuracy of the model on the dataset.

8 Conclusion

This application note provided an introduction into transfer learning and explained the importance of having the correct dataset for a given machine learning problem. It focused on improving the performance of the handwritten digit recognition application presented in AN12603. Significant improvement was achieved by collecting a new dataset consisting of inputs gathered through the same method that they are taken in the digit recognition application and then performing transfer learning from the original MNIST-trained model. The benefits of transfer learning were measured through several experiments presented in Results.

9 References

- Dongmei, H., Qigang, L., & Weiguo, F. (2018). A new image classification method using CNN transfer learning and web data augmentation. *Expert Systems with Applications, 95*, pp. 43-56. doi:10.1016/j.eswa.2017.11.028
- Fard, F. S., Hollensen, P., Mcilory, S., & Trappenberg, T. (2017). Impact of biased mislabeling on learning with deep networks. *2017 International Joint Conference on Neural Networks (IJCNN)*, (pp. 2652-2657). Anchorage. doi:10.1109/IJCNN.2017.7966180

- Ferreira et al., .. (2018). Classification of Breast Cancer Histology Images Through Transfer Learning Using a Pre-trained Inception Resnet V2. *Lecture Notes in Computer Science, 10882*. doi:10.1007/978-3-319-93000-8_86
- Google. (2020). *Quick, Draw! The Data*. Retrieved from https://quickdraw.withgoogle.com/data
- Graetz, F. M. (2019, January). *How to visualize convolutional features in 40 lines of code*. Retrieved from towardsdatascience: https://towardsdatascience.com/how-to-visualize-convolutional-features-in-40-lines-of-code-70b7d87b0030
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012, 1). ImageNet Classification with Deep Convolutional. *Advances in neural information processing systems*. doi:10.1145/3065386
- Ng, A. Y. (2016). *Nuts and Bolts of Building Applications using Deep Learning*. Retrieved from Neural Information Processing Systems Conference: https://channel9.msdn.com/Events/Neural-Information-Processing-Systems-Conference/Neural-Information-Processing-Systems-Conference-NIPS-2016/Nuts-and-Bolts-of-Building-Applications-using-Deep-Learning
- NXP Semiconductors. (2019). *eIQ Transfer Learning Lab with i.MX 8*. Retrieved from NXP Community: https://community.nxp.com/docs/DOC-343848
- NXP Semiconductors. (2019). *eIQ Transfer Learning Lab with i.MX RT*. Retrieved from NXP Community: https://community.nxp.com/docs/DOC-343827
- NXP Semiconductors. (2019). *Using a TensorFlow Lite Model to Perform Handwritten Digit Recognition on RT1060*. Retrieved from NXP EIQ Community: https://community.nxp.com/docs/DOC-345190
- Plesa, N. (2020). *Machine learning datasets*. Retrieved from datasetlist: https://www.datasetlist.com/
- Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). *"Why Should I Trust You?": Explaining the Predictions of Any Classifier.* Retrieved from arXiv: https://arxiv.org/abs/1602.04938
- Ruder, S. (2016, 12 21). *Highlights of NIPS 2016: Adversarial learning, Meta-learning, and more*. Retrieved from https://ruder.io/highlights-nips-2016/
- Sanderson, G. (2018). Neural networks. YouTube. Retrieved May 28, 2020, from https://youtu.be/aircAruvnKk
- Sarkar, D. (2018). *A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning*. Retrieved May 28, 2020, from towardsdatascience: https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a
- Shaopeng, L., Guohui, T., & Yuan, X. (2019). A novel scene classification model combining ResNet based transfer learning and data augmentation with a filter. *Neurocomputing, 338*, 191-206. doi:10.1016/j.neucom.2019.01.090

A Neural network basics

A neural network model consists of a network and its weights and biases. The network represents the structure of the model – how many layers the model has, how many neurons there are in each layer and the connections between neurons of different layers. The first layer's neurons take values directly from the input data. For example, each neuron in the first layer might receive the greyscale value of a single pixel from an image. A neuron is a mathematical function, which has a set of inputs, a weight for each of the inputs, a bias, an activation function and an output. The inputs are values from preceding layers (or the input data). The weights assign importance to each input, so that the neuron can pick which inputs it focuses on. The bias further adjusts the neuron's result. The activation function converts linear input into non-linear output, so that the model as a whole can look for complicated patterns in data. The output is the result of a sum of all inputs multiplied with their weights, plus the neuron's bias and that result is mapped with the activation function. Equation 1 shows the whole layer's equation, where σ is an activation function, w_i is a matrix of weights, a_i is a vector of neurons, and b_i is a vector of biases. The vector a_i contains the outputs of the layer i and each row of the matrix w_i contains the weights that a single neuron from a_{i+1} assigns to the neurons from a_i.

$$a_i + 1 = \sigma\,(w_i{}^* a_i + b_i)$$

Equation 1. Layer equation

It is through these weights and biases as well as the neurons layouts and layer topologies that the model looks for patterns in data. When a neuron is set to look for a certain pattern, its weights are high for the inputs that would represent this pattern and low for all the other inputs. When the combined input of a neuron contains the searched pattern, the neuron‘s output is high. When the input does not contain the pattern, the neuron’s output is low.The better the combined input matches a neuron’s searched pattern, the higher its output value, and usually this relationship is non-linear.

When it comes to a classification network, the last layer of the network decides what class the model thinks the model’s input belongs to. This is the result of all of the computations of the previous layers. Each neuron of the last layer represents a different class and has a value that states the possibility of the input belonging to that specific class.

When the model is training, it adjusts the weights and biases based on how accurately it guessed the output value or class for the input data. But the neurons layout and layer topologies of the model, which are often called **hyper parameters**, determined by model’s architecture, are not changed by training. Training is divided into steps and epochs. During each step, the network takes a batch of examples, the amount of examples is determined by the chosen batch size, runs inference on all of them and then adjusts the weights and biases based on how well it performed. Performance is measured as an error rate between the inference result and the expected output. In one epoch, the model runs enough steps to go through the whole training dataset. The amount of epochs used for the training can be experimented with, and the more epochs the model is given, the more accurate it usually becomes, with diminishing returns. The model can also run into issues, which causes it to get worse with further epochs. This can be solved by either stopping the training before the model accuracy starts deteriorating or changing the model’s architecture or dataset.

As usual with machine learning and AI, the topic of neural networks has been described in various detail and complexity by third-party sources. Among these is a YouTube series that explains neural networks from the basics to the more complex mechanisms, which makes it very useful for beginners. It was made by the channel 3Blue1Brown (Sanderson, 2018).

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 06/2020

Document identifier: AN12892