Heat Conduction Modeling and Simulation using Deep Learning Techniques

April 23, 2024

Autor: Ondřej Šperl

Abstract: The present work is devoted to the surrogate modeling of heat conduction problem using deep neural networks. Deep learning is a subfield of machine learning that employs artificial neural networks to extract intricate patterns and relationships from data. These networks consist of multiple layers of interconnected nodes, which enable the automatic learning of hierarchical representations of data. Unlike traditional machine learning algorithms that require handcrafted features, deep learning algorithms can learn features directly from raw data, making them highly effective in tasks such as image and speech recognition, natural language processing, and predictive analytics. In this study, the U-Net convolutional neural network is utilized to model stationary heat transfer. The U-Net is a powerful deep learning architecture that has been widely used in image segmentation tasks, and its application to heat transfer modeling represents a novel approach to this important problem in engineering and physics.

Keywords: Heat conduction, Deep learning, U-Net architecture

1 Heat conduction

To begin with, we introduce a bounded body $D \subset R^3$ (reference configuration) with a piecewise smooth boundary Γ. In particular, the Dirichlet, Neumann, and Robin boundary conditions are imposed on $\Gamma_D \subset \Gamma$, $\Gamma_N \subset \Gamma$ and $\Gamma_R \subset \Gamma$, such that $\Gamma = \Gamma_D \cup \Gamma_N \cup \Gamma_R$. Moreover, to investigate the time-dependent behavior of D, thus we consider a time interval $[0, t_s] \subset R_+$. The evolution of temperature in D is expressed as

$$\theta : D \times [0, t_s] \longrightarrow R^3, \tag{1}$$

where $\theta\,[°\mathrm{C}]$ is the temperature. Heat transport is then described by the transient heat balance equation with initial and boundary conditions as

$$\begin{cases} c_v(x)\dfrac{\partial \theta}{\partial t}(x,t) - \nabla \cdot (\lambda(x)\nabla\theta(x,t)) = 0, & x \in D,\ t \in (0,t_s)\,, \\ \theta(x,t) = \theta_D(x,t), & x \in \Gamma_D,\ t \in (0,t_s)\,, \\ \lambda(x)\dfrac{\partial \theta}{\partial n}(x,t) = q_N(x,t), & x \in \Gamma_N,\ t \in (0,t_s)\,, \\ \alpha(\theta(x,t) - \theta_\infty(x,t)) = \lambda(x)\dfrac{\partial \theta}{\partial n}(x,t), & x \in \Gamma_R,\ t \in (0,t_s)\,, \\ \theta(x,0) = \theta_{in}(x), & x \in D. \end{cases} \tag{2}$$

where $c_v(x)$ $[\mathrm{Jm}^{-3}\mathrm{K}^{-1}]$ is the volumetric heat capacity introduced as a product of the volumetric mass density $\rho_s(x)\,[\mathrm{kgm}^{-3}]$ and the specific heat capacity $c_p(x)\,[\mathrm{JKg}^{-1}\mathrm{K}^{-1}]$, i.e., $c_v(x) = \rho_s(x)c_p(x)$, $\lambda(x)\,[\mathrm{Wm}^{-1}\mathrm{K}^{-1}]$ is the thermal conductivity, $t_s\,[\mathrm{s}]$ is the final time of the simulation, $\theta_\infty(x,t)$ $[°\mathrm{C}]$ is the ambient temperature, α $[\mathrm{Wm}^{-2}\mathrm{K}^{-1}]$ is the heat transfer coefficient, $\theta_D(x,t)$ $[°\mathrm{C}]$ is the prescribed temperature and $q_N\,[\mathrm{Wm}^{-2}]$ is the prescribed heat flux. As a preamble, we limit ourselves to the stationary problem. This simplification offers a more tractable approach to data handling during the training phase of neural networks, thereby allowing for a more thorough and in-depth analysis of the problem at hand.

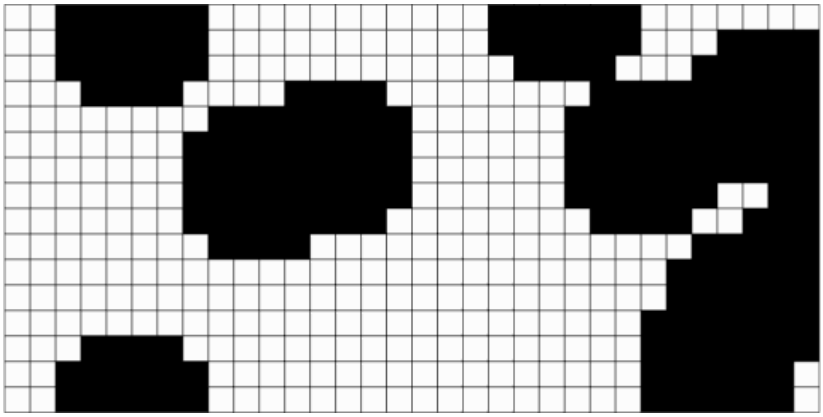
[1]:
```
from Unet_functions import *
import tensorflow as tf
from scipy.io import loadmat
import pandas as pd
from io import StringIO
import math
from Graphs import *
import os
```

2 Finite element discretization and dataset preparation

The finite element method (FEM) represents a systematic technique for the approximation of solutions to partial differential equations (PDEs) by projecting the infinite-dimensional function space onto a finite-dimensional subspace. This process involves the conversion of continuous functions into discrete functions, which are then represented as ordinary vectors within a vector space. These vectors can be effectively manipulated using numerical methods. One widely used formulation of the finite element method is the Galerkin method, which is employed for the discretization process. The Galerkin method is based on the principle of weighted residuals, where the residual is orthogonal

to the chosen finite-dimensional subspace. This method ensures that the error between the exact solution and the approximate solution is minimized in a well-defined mathematical sense.

In this study, a 2D rectangular domain subjected to Dirichlet boundary conditions on its left and right sides is discretized utilizing quadrilateral finite elements. Each quadrilateral element corresponds to a single pixel, which represents the given material phase. The investigated domain comprises two material phases with distinct thermal conductivities, i.e. $\lambda_1 = 1.0\,\mathrm{Wm^{-1}K^{-1}}$ and $\lambda_2 = 5.0\,\mathrm{Wm^{-1}K^{-1}}$. To account for the heterogeneity of the material, we model it by randomly positioning overlapping circles characterized by λ_1 within the domain predominantly characterized by λ_2, see attached figure below. The successful training of neural networks necessitates the availability



of a well-prepared dataset that enables efficient learning and guarantees optimal performance. In this context, a total of 20,000 finite element simulations were performed, utilizing randomly generated input domains to encompass the essential features required for comprehensive learning. This extensive dataset ensures the neural network's ability to accurately model complex relationships and generalize well across various scenarios.

[2]:
```
path = "..\InputsForDNN\MESH_Rec_15_x_31_1\\"
grid = "..\Grids\\MESH_Rec_15_x_31_1.mat"
model_database_dir = "..\\U_Net\Models\\" # directory where trained models are␣
↪stored
```

[3]:
```
lambdas = np.array(loadmat(path+"Lambdas.mat")["Lambdas"])
temperatures = np.array(loadmat(path+"NodeTemperatures.mat")["Temperatures"])
elementTemperatures = np.array(loadmat(path+"ElementTemperatures.
↪mat")["ElementTemperatures"])
nodeLambdasAsCodeNumbers = np.array(loadmat(path+"NodeLambdasAsCodeNumbers.
↪mat")["NodeLambdas"])

lambdas = lambdas.reshape((20000, 15, 31))
elementTemperatures = elementTemperatures.reshape(20000, 15, 31)
temperatures = temperatures.reshape(20000, 16, 32)
nodeLambdasAsCodeNumbers = nodeLambdasAsCodeNumbers.reshape(20000, 16, 32)
```

3 Deep Learning

Here, It is essential to introduce the context between artificial intelligence (AI), machine learning, and deep learning. AI refers to the development of machines and algorithms that can perform tasks that typically require human intelligence, such as recognizing patterns or making decisions. Machine learning is a subset of AI, which enables machines to learn from data without being explicitly programmed. Machine learning algorithms can analyze large datasets, such as those used for playing chess, or identify patterns and relationships within the data. Deep learning is a subset of machine learning that uses artificial neural networks with multiple layers to model and solve complex problems. By leveraging the power of neural networks, deep learning has achieved state-of-the-art results in a wide range of applications. For that reason, all proposed solutions in this work are based on deep learning bringing the ready-made, well-established, and well-documented software libraries, testing and benchmarking methodologies such as underfitting, overfitting, and cross-validation, and standards for measuring model performance.

[4]:
```
Training_data = 15000
Validation_data= 2500
Testing_data = len(lambdas) - Validation_data - Training_data
```

Data splitting into training, validation, and testing sets is crucial for several interconnected reasons that contribute to the development of accurate and reliable deep learning models. This practice allows for an unbiased evaluation of the model's performance on unseen data, preventing overfitting by monitoring the model during the training process and detecting any signs of learning the training data too well, including noise and outliers. The validation set plays a vital role in fine-tuning hyperparameters, such as learning rate, batch size, or the number of hidden layers in a neural network, and in comparing multiple models or algorithms to determine which one performs best on unseen data. By using separate datasets for training, validation, and testing, we ensure that the model's performance is estimated without bias, which is crucial for obtaining reliable and accurate results, especially in real-world applications where the model's performance on new data is of utmost importance. From mentioned reasons, the dataset is here partitioned into three distinct subsets: the training dataset, the validation dataset, and the testing dataset, with the following respective sizes: 15,000, 2,500 and 2,500.

4 Data normalization

Data normalization is a vital preprocessing step in the context of neural networks for several reasons: i) It speeds up the learning process as gradient-based learning algorithms, typically used in neural networks, perform optimally when input features are on a similar scale. If one feature has a much larger range than others, the algorithm's iteration through that feature's values slows down, hindering the learning process. ii) Normalizing data can help avoid the problem of local minima, where neural networks often get stuck during optimization. By smoothing and easing the optimization landscape, the network can navigate better. iii) The performance of the model can be significantly improved as neural networks frequently use activation functions sensitive to the scale of their inputs. Functions like sigmoid and tanh can saturate if inputs are too large or small, causing slow or poor learning. iv) Normalization can also reduce overfitting by helping the model generalize better and be less sensitive to specific input feature values. iv) It provides numerical stability by preventing issues caused by very large or small numbers, leading to more accurate computations and results.

[4]:
```
# Normalizing input and output
Lambdas = np.unique(nodeLambdasAsCodeNumbers)
MaxLambda = Lambdas.max()
MinLambda = Lambdas.min()
Inputs = (nodeLambdasAsCodeNumbers - MinLambda)/(MaxLambda - MinLambda)

MaxTemperature = temperatures.max()
MinTemperature = temperatures.min()
Outputs = (temperatures - MinTemperature)/(MaxTemperature - MinTemperature)

TrainingInputs = Inputs[:Training_data]
ValidationInputs = Inputs[Training_data:Training_data+Validation_data]
TestingInputs = Inputs[Training_data+Validation_data:]

TrainingOutputs = Outputs[:Training_data]
ValidationOutputs = Outputs[Training_data:Training_data+Validation_data]
TestingOutputs = Outputs[Training_data+Validation_data:]
print(TrainingOutputs.shape)
print(TestingInputs.shape)
```

[5]:
```
try:
    ModelsDataFrame = pd.read_csv(model_database_dir+"ModelsDatabase.csv")
    NewIndex = ModelsDataFrame.index[-1] + 1
except:
    ModelsDataFrame = pd.DataFrame({})
    NewIndex = 1

model_path = model_database_dir+"Model"+str(NewIndex)

if not os.path.exists(model_path):
    os.makedirs(model_path)

model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=model_path,
    monitor='val_loss',
    mode='min',
    save_best_only=True)
```

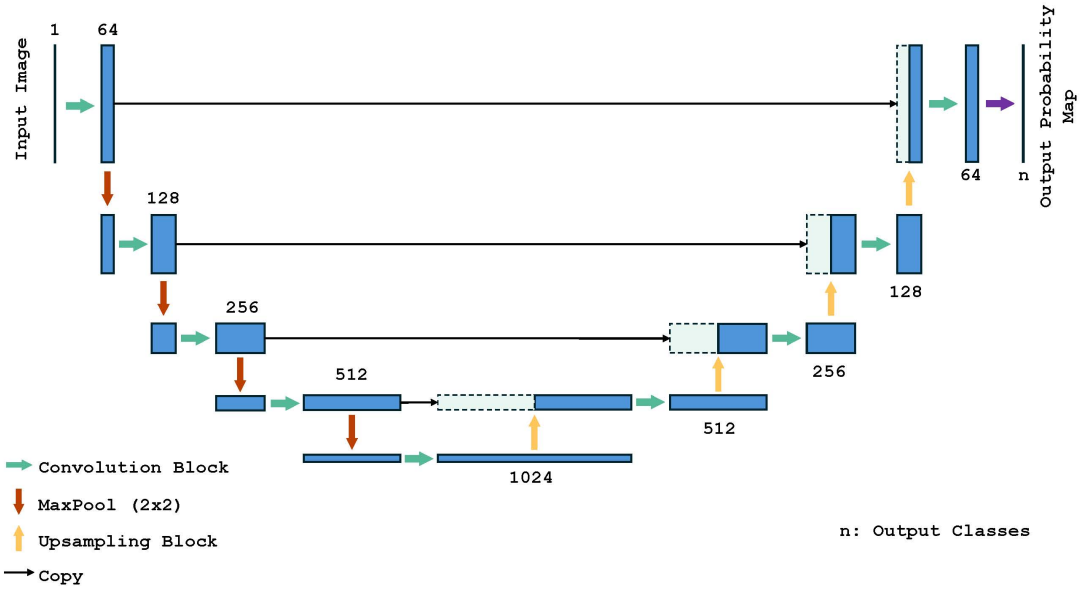
5 U-Net Architecture

The U-Net architecture, introduced by Ronneberger et al in 2015, is derived from the "fully convolutional network" introduced by Long, Shelhamer, and Darrell in 2014. The core concept involves enhancing a typical contracting network with subsequent layers, where traditional pooling operations are substituted with upsampling operators. These layers increase the output resolution, enabling successive convolutional layers to generate a precise output based on the provided information. A significant adjustment in the U-Net architecture is the incorporation of a large number of feature channels in the upsampling section, facilitating the propagation of context information to higher

resolution layers. Consequently, the expansive path exhibits near symmetry with the contracting part, resulting in the distinctive u-shaped architecture.

[6]:
```
def build_unet_model():
    # building model
    # inputs
    inputs = layers.Input(shape=(16,32,1))


    f1, p1 = downsample_block(inputs, 64)
    f2, p2 = downsample_block(p1, 128)
    f3, p3 = downsample_block(p2, 256)

    bottleneck = double_conv_block(p3, 512)

    u1 = upsample_block(bottleneck, f3, 256)
    u2 = upsample_block(u1, f2, 128)
    u3 = upsample_block(u2, f1, 64)


    # outputs
    outputs = layers.Conv2D(1, 1, padding="same", activation = "linear")(u3)

    # unet model with Keras Functional API
    unet_model = tf.keras.Model(inputs, outputs, name="U-Net")
    return unet_model
```

Typical U-Net architecture is depicted in the following figure (courtesy of Kaustav Das).



[7]:
```
unet_model = build_unet_model()
unet_model.summary()
```

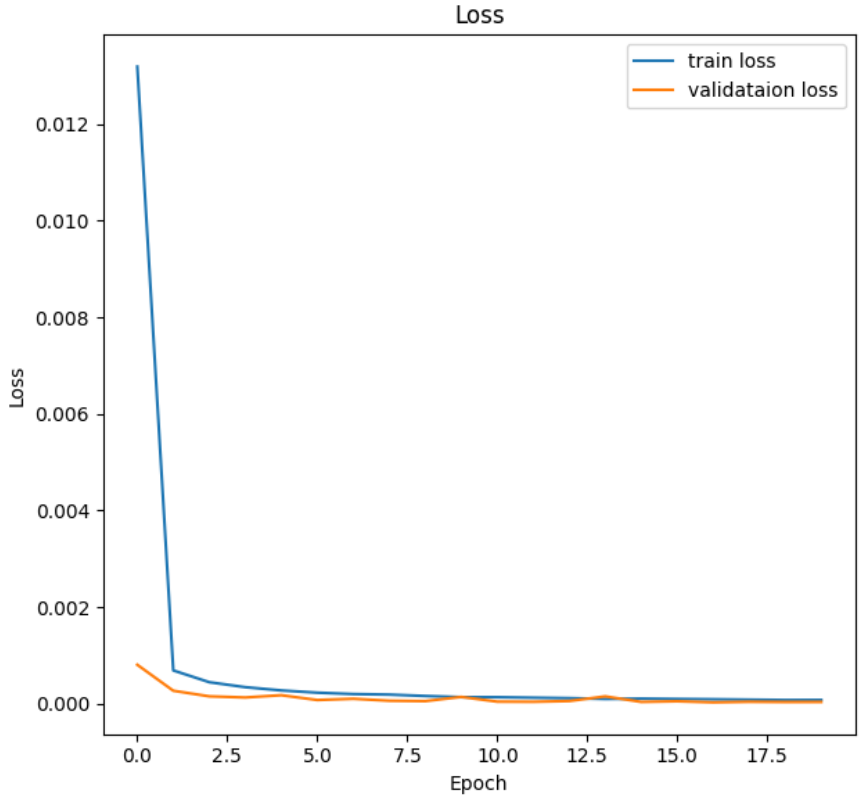
The total quantity of unknown weights, which require training, exceeds 8 million. Optimization algorithms are indispensable components in the domain of deep learning, as they enable neural networks to learn with high efficiency and converge towards optimal solutions. One of the most popular optimization techniques used in training deep neural networks is the Adam optimizer, which is also utilized for our training case.

[8]:
```
unet_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                   loss="mean_squared_error")
```

[9]:
```
epochs = 20
model_history = unet_model.fit(TrainingInputs,
                               TrainingOutputs,
                               epochs=epochs,
                               validation_data=[ValidationInputs,␣
↪ValidationOutputs],
                               callbacks=[model_checkpoint_callback])
```

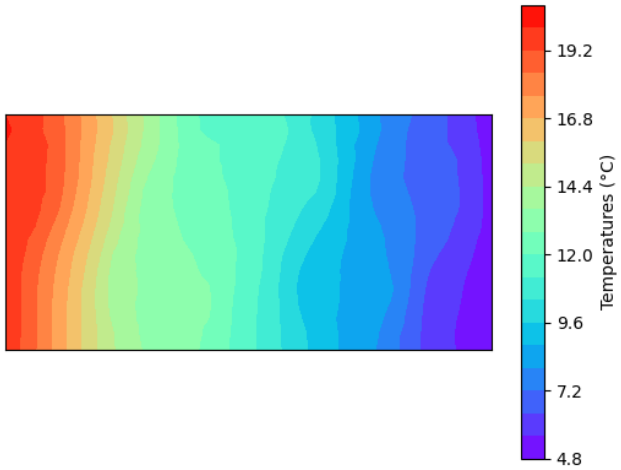
[11]:
```
display_learning_curves(model_history.history)
```

The evolutions of training loss and validation loss are depicted in the following figure.

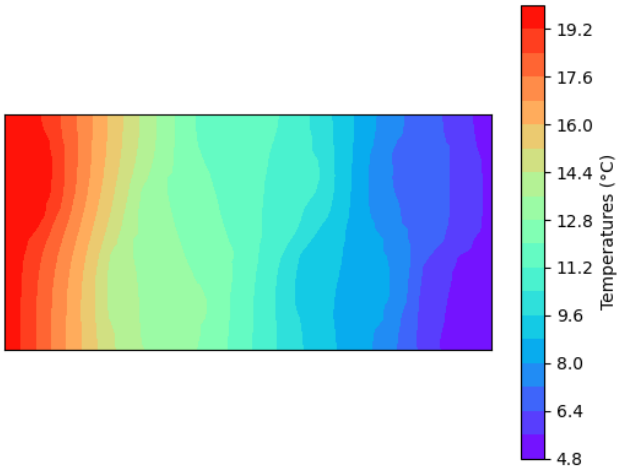


[12]:
```
unet_model = tf.keras.models.load_model(model_path)
etaps_of_prediction = math.floor(Testing_data / 2500) # Splitting testing␣
↪dataset due to GPU memory - I can not load big tensors to GPU.
predictions = np.zeros((Testing_data, 16, 32))
for i in range(etaps_of_prediction):
    prediction_part_i = unet_model(TestingInputs[i*2500: (i+1)*2500].
↪reshape((2500, 16, 32, 1)))
    prediction_part_i = tf.reshape(prediction_part_i, (2500, 16, 32)).numpy()
    predictions[i*2500: (i+1)*2500] = prediction_part_i

RemainingData = Testing_data - etaps_of_prediction * 2500
prediction = unet_model(TestingInputs[etaps_of_prediction*2500: ].
↪reshape((RemainingData, 16, 32, 1)))
prediction = tf.reshape(prediction, (RemainingData, 16, 32)).numpy()
predictions[etaps_of_prediction*2500:] = prediction
```

The performance of the proposed technique is observed in the comparison of FEM-predicted and Deep learning-based temperature fields computed for one particular material domain illustrated in the following figures.



FEM-predicted field



Deep learning-based field

6 Conclusions

We briefly present the use of deep neural networks in the problem of heat conduction. We are aware of the length and quality of the manuscript, thus the oral presentation is an integral part of the studied topic.