Last updated: January 10, 2023

Bonus Points Assignment II

FUNDAMENTALS OF MACHINE LEARNING

Prof. Dr. Sebastian Trimpe

Introduction

In this bonus points assignment, you will implement a neural network in Python to solve a classification problem on an industrial data set. Namely, our goal is to detect defects on pictures of a part of hydraulic pumps, called the submersible pump impeller (Fig. 1). This is one of the applications of machine learning in industry to automate time-consuming manual processes. Throughout the assignment, you will discover how to use PyTorch, work with state-of-the-art pre-trained networks, and fine tune them for your own task.

The assignment is divided as follows. We begin by implementing a neural network from scratch in NumPy. The goal is to firmly establish your understanding of the lecture by translating it into tangible code. In the second part, we use the library PyTorch to define and train a convolutional neural network (CNN). We apply this to the problem at hand in the last part, where we fine-tune the state-of-the-art model VGG to solve our classification task. The three parts can be solved independently.

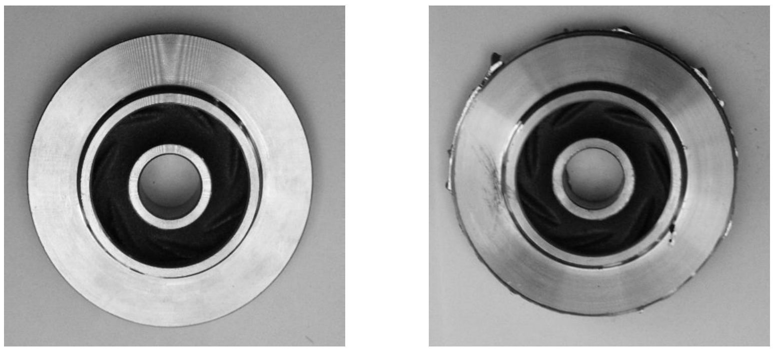

Figure 1: Submersible pump impellers, without (left) and with (right) manufacturing defects.

Logistics

Formalia This voluntary assignment can earn you up to 5% of bonus points for the final exam. The bonus points will only be applied if you pass the exam without them; a failing grade cannot be improved with bonus points. This assignment starts on *January 9th* and is due on *January 30th*. It has to be submitted in groups of 1 to 4 students. All group members have to register in the same group on Moodle under *Groups Assignment II*. Upload the jupyter notebooks on Moodle under the appropriate assignment. Do not change the names of the notebooks, and do not clear their outputs.

Grading Grading will involve automated tests, so **do not change predefined functions or variable names** as some of the tests depend on them. Please make sure your solutions run correctly without raising exceptions. The lines with `assert` statements are tests for you to check the outcome of your code.

Allowed Libraries The libraries you are allowed to use change as you progess through the assignment. We always indicate clearly what these allowed libraries are, and you can always use built-in Python functionalities. The purpose is to have you implement most algorithms and functions by hand once so you have a clear idea of what is hiding behind them. Then, you use higher-performance implementations for applications. You are not allowed to use any library that is not already installed in the virtual environment we provide.

Jupyter You may use the RWTH JupyterHub to run and edit Jupyter Notebooks(profile “[FML] Fundamentals of Machine Learning”). Alternatively, you can set up your environment locally, as explained on Moodle, or use Google Colab for collaborative work. Due to resource constraints, we will not provide support for alternatives to the JupyterHub.

Please report any mistakes found in this assignment to the teaching assistants, on the appropriate Moodle forums or via email (fml@dsme.rwth-aachen.de).

1 Neural Networks from Scratch

In this section, we implement neural networks from scratch in NumPy. We particularly focus on *feedforward fully-connected* networks, that is, the kind that was covered in details in the lecture. We also implement a convolution, the basic element of CNNs, but do not implement a full such network as this would be unnecessarily difficult.

Allowed Libraries In this section, you may only use `numpy`.

Setup

We adopt a modular architecture similar to that of PyTorch but simplified to suit our needs. We represent the layers of the neural network by instances of the base class `Layer`. This class defines three abstract methods:

- `def forward(self, x)`: this is the forward pass of the layer. It receives the output of a previous layer (or the input), applies its transformation to the data and returns the output of the layer.
- `def backward(self, gradient)`: this is the backward pass of the layer. It receives the gradient of the following layer (or of the loss w.r.t. the output of the network), updates it, and returns this new gradient so it is passed to the preceding layer.
- `def update(self, learn_rate)`: this method updates the weights of the layer according to the learning rate and the previously-computed gradients.

The layers are then subclasses of `Layer` and implement these three methods.

Finally, all layers support *batched input*. That means that the first dimension of the input (either `x` or `gradient`) is used to handle multiple examples. For instance, if the network processes vectors of dimension D and outputs vectors of dimension K, then the input and output array of the network has shape `(B, D)` and `(B, K)`, respectively, where B is the batch size and can be set arbitrarily. The output then has the shape

Implementing a Linear Layer

The first layer we implement is the linear layer. It is defined by the input-output equation

$$x_{\text{out}} = x_{\text{in}}^\top \cdot w + b, \tag{1}$$

where x_{in} is the input, x_{out} is the output, and b is the bias.

1.1. Implement the class `MyLinearLayer`.

(a) Implement the method `forward`. It takes in a batch of data `x` and performs the transformation (1) using the weights `w` and bias `b` defined in the class constructor.

(b) Implement the method `backward`. It takes in a batch of gradients `gradient` with respect to the output of the layer and computes the gradient with respect to the inputs of the layer.

(c) Implement the method `update`. It takes in a learning rate, computes the gradients of the loss w.r.t. the parameters `w` and `b`, and performs a gradient descent step.

Hint. *You need to store the input in the `forward` method and the incoming gradient in the `backward` method.*

Implementing ReLU

Our network will use the ReLU activation function, defined as

$$x_{\text{out}} = \max(0, x_{\text{in}}), \tag{2}$$

where x_{in} is the input, x_{out} is the output, 0 is the matrix full of 0's of the same shape as x_{in}, and max is the element-wise maximum. This layer does not have any trainable parameters; therefore, its `update` method does not do anything.

1.2. Implement the class `MyReLU`.

   (a) Implement the method `forward`. It takes in a batch of data `x` and performs the transformation (2).

   (b) Implement the method `backward`. It takes in a batch of gradients `gradient` with respect to the output of the layer and computes the gradient with respect to the inputs of the layer.

   Hint. *The* ReLU *function is not differentiable at* 0*. Choose an appropriate value for the gradient.*

Putting it together

We are now ready to combine these layers to implement a neural network. We do this in the class `MyNeuralNet`, which defines a network with one hidden layer and one output neuron. The hidden layer has ReLU activation, and the output layer has linear activation. This class implements the methods `forward`, `backward`, and `update`; they respectively call the corresponding methods of the different `Layer` objects that compose the network. We limit ourselves to one-dimensional regression with the least-squares loss function

$$L(w) = \frac{1}{2N} \sum_{n=1}^{N} (\mathbf{y}_n - h(\mathbf{x}_n, w))^2, \tag{3}$$

where $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N) \in \mathbb{R}^2$ are data points and h is the neural network. This definition of the loss slightly differs from the one seen in the lecture; we normalize the sum with the number of data points. This is useful to compare the loss of the network evaluated on data sets of different sizes. The gradient of the loss w.r.t. the output of the network is

$$\frac{\partial L}{\partial h} = \frac{1}{N} \sum_{n=1}^{N} h(\mathbf{x}_n, w) - \mathbf{y}_n. \tag{4}$$

1.3. Implement the class `MyNeuralNet`.

   (a) Implement the method `forward`. It takes in a batch of data `x` and feeds it to the `forward` method of the different layers successively.

   (b) Implement the method `backward`. It performs backpropagation by first computing the gradient of the loss w.r.t. the most recent output of the network as per (4), and then successively passing it through the `backward` methods of the layers.

   (c) Implement the method `update`. It calls the `update` method of all the layers.

1.4. Fill in the function `train_myneuralnet`. It takes as input an untrained network of the class `MyNeuralNet`, a number of epochs, a batch size, a learning rate, and a number of training points. It trains the network on data given by the function `get_training_data_myneuralnet` (already implemented).

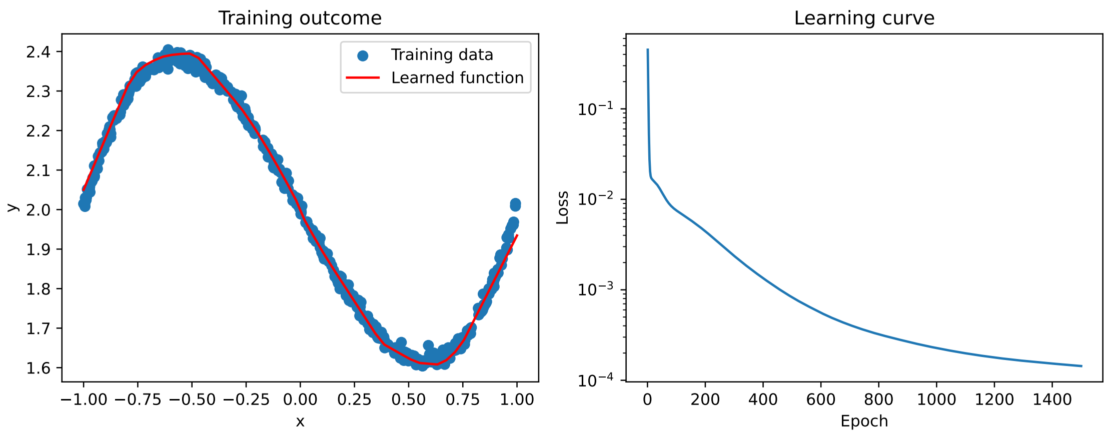

Figure 2: The outcome of training `MyNeuralNet`.

1.5. Call the function `train_and_plot_myneuralnet` to train the network and plot the learned function. Use `n_hidden=200, epochs=1500, batch_size=4, learn_rate=0.01`, and `N=500`. You should get an output similar to Figure 2.

This concludes the part on implementing a feedforward fully-connected network. You can play around with the function `train_and_plot_myneuralnet` by changing the values of the parameters and seeing how this affects training.

Implementing Convolutions

Another important class of networks is that of CNNs. They work by chaining convolutions, pooling, and activation functions. In this part, we implement such a convolution operation. To match the usual use case of CNNs, the input to this convolution is an image.

Images and Kernels We represent an image as an `np.ndarray` of shape `(B, h, w, F)`, where `h` and `w` are respectively the height and width of the image in pixels and `F` is its number of filters. An RGB image has 3 filters, each corresponding respectively to red, green, and blue. We support batched inputs, and `B` is the batch size.

Example. *If `images` is a batch of RGB images, then `images[b, :, :, 1]` is a 2D array containing all pixels of the green filter of image number $b \leq B - 1$.*

We convolve such images with *kernels*. A kernel is an `np.ndarray` of shape `(K, K, F_in, F_out)`, where `K` is the kernel height and width, and `F_in` and `F_out` are respectively the number of input and output filters. For simplicity, we only consider square kernels.

Padding Images can be padded with 0 values on their side. We assume that the same amount of padding is added on all sides and filters of the image; we denote by P the amount of padding on one side. Padding, therefore, increases the size of the image in each dimension by $2 \cdot P$.

Convolution Given a padded input image $(x_{s,t}^{f_{\text{in}}})$ of shape $(h_{\text{in}} + 2P, w_{\text{in}} + 2P, F_{\text{in}})$, we transform it into an output image $(z_{p,q}^{f_{\text{in}}})$ of shape $(h_{\text{out}}, w_{\text{out}}, F_{\text{out}})$ with the *convolution operation* defined as:

$$z_{p,q}^{f_{\text{out}}} = \sum_{f_{\text{in}}=0}^{F_{\text{in}}-1} \sum_{i,j=0}^{K-1} k_{i,j}^{f_{\text{in}},f_{\text{out}}} \cdot x_{p\cdot S+i,\ q\cdot S+j}^{f_{\text{in}}} \,, \tag{5}$$

Identity	Sharpening	Edge Detection 1
$\begin{pmatrix}0&0&0\\0&1&0\\0&0&0\end{pmatrix}$	$\begin{pmatrix}0&-1&0\\-1&5&-1\\0&-1&0\end{pmatrix}$	$\begin{pmatrix}1&0&-1\\0&0&0\\-1&0&1\end{pmatrix}$
Edge Detection 2	**Edge Detection 3**	**Gaussian Blur**
$\begin{pmatrix}0&1&0\\1&-4&1\\0&1&0\end{pmatrix}$	$\begin{pmatrix}-1&-1&-1\\-1&8&-1\\-1&-1&-1\end{pmatrix}$	$\frac{1}{16}\begin{pmatrix}1&2&1\\2&4&2\\1&2&1\end{pmatrix}$

Table 1: 3×3 kernels commonly used in classical image processing. They are implemented in most image editing software.

where S is the stride, and for all $(p, q, f_{\text{out}}) \in \{0, \dots, h_{\text{out}}-1\} \times \{0, \dots, w_{\text{out}}-1\} \times \{0, \dots, F_{\text{out}}-1\}$. We reserve the indices i and j for indexing over the kernel, p and q for indexing over the output image, and s and t for indexing over the input image. For this formula to be meaningful, we need the following relation between the different dimensions involved:

$$h_{\text{out}} = \frac{h_{\text{in}} + 2\cdot P - K}{S} + 1, \tag{6}$$

$$w_{\text{out}} = \frac{w_{\text{in}} + 2\cdot P - K}{S} + 1. \tag{7}$$

1.6. Implement the function `add_padding`. It takes in a batch of images `images` and the number `padding` of padding neurons to add.

   Hint. *Use the function* `np.pad` *with the correct arguments.*

1.7. Implement the function `convolve`, which implements a convolution as per (5). It takes in the kernel `kernels`, a batch of `images`, and a positive stride `stride`. Assume the image is already padded, that is, do *not* add padding in this function.

   Hint. *Disregard code efficiency and use* `for` *loops.*

Convolutions are central in image processing. A good choice of the kernel can achieve operations as varied as sharpening, edge detection, or blurring. You can find in Table 1 famous kernels that achieve these operations.

1.8. Perform sharpening, edge detection, and blurring of the example image by running the function `filter_example`. You should get an output similar to Figure 3.

In the previous example, we applied *given* kernels to the input image. This contrasts with the usual use case of CNNs where the kernels are *learned* by optimizing a loss function; this automatic tuning is one of the main contributions of CNNs. The backpropagation algorithm is key here, since it enables automatic and efficient computations of the gradients. Implementing it for CNNs is, however, notoriously difficult; that is why we use specialized high-performance libraries such as PyTorch.

2 Defining and Training Models in PyTorch

In this section, we use the state-of-the-art library PyTorch to build and train a CNN. We use this network to solve an image classification task for defect detection on industrially manufactured parts [2].

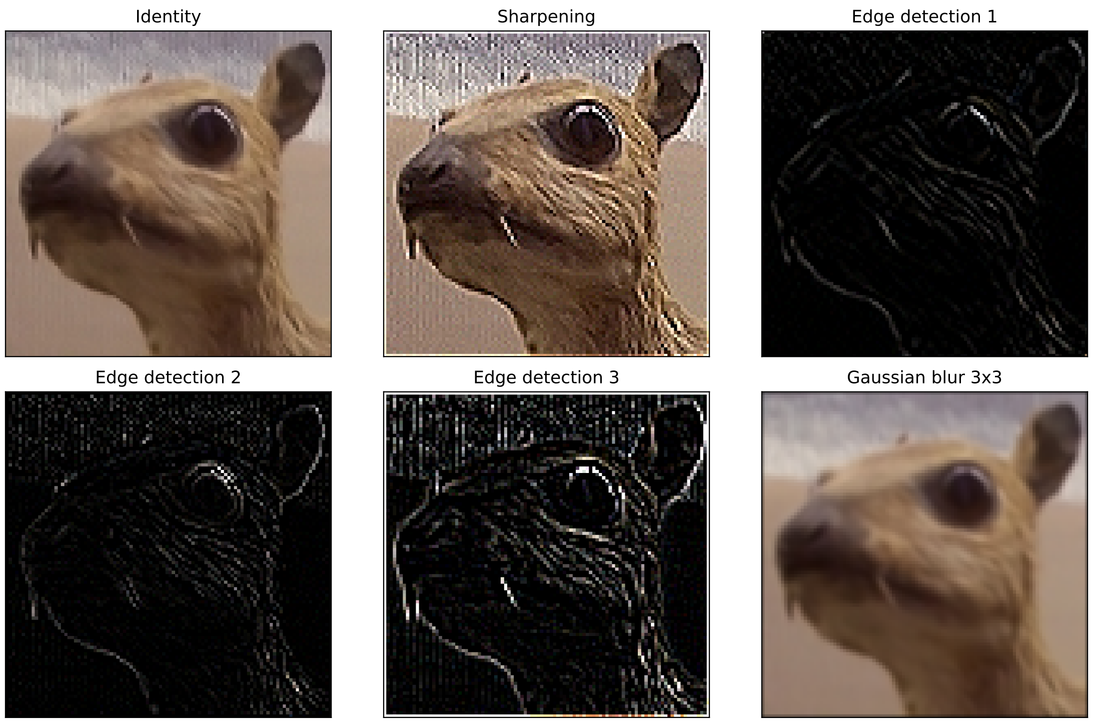

Figure 3: The result of convolving the example image with the kernels of Table 1. Each kernel has a very specific action on the input image, such as sharpening, blurring, or edge detection. Source of the original image: [1].

Allowed Libraries In this section, you may use `torch`, `numpy`, and `matplotlib`.

Defining a CNN in PyTorch

A PyTorch crash-course PyTorch is a library that enables users to easily and quickly train neural networks in Python. Its fundamental object is called a `Tensor`. Tensors are extremely similar to NumPy arrays, with two notable additions:

- Operations on tensors can be performed by the GPU;
- Operations on tensors implement automatic differentiation.

This second point is core to training neural networks, as it enables seamlessly performing back-propagation. You can read more on this in the PyTorch documentation [3].

The second main component of PyTorch is the `nn` module. It contains the classes and functions necessary to build a neural network. A neural network is then a subclass of the class `nn.Module`, and implements a `forward` method that computes the output of the network by chaining operations on tensors. Unlike our implementation of the previous section, PyTorch automatically implements the `backward` method. Relevant for this section are the classes `nn.Linear`, `nn.Conv2d`, `nn.Flatten`, `nn.MaxPool2d`, `nn.ReLU`, and `nn.Sigmoid`. You can find out how to use them in the PyTorch documentation [4].

2.1. Implement in PyTorch the CNN `SimpLeNet`, whose successive layers are:

1. a convolutional layer with $F_{\text{in}} = 3,\ F_{\text{out}} = 6,\ K = 3,\ P = 1$, and $S = 4$;

2. a max-pooling layer with $K = 4$ and $S = 2$;
3. a flattening layer;
4. a linear layer with 10 output neurons;
5. a ReLU activation;
6. another linear layer with 1 output neuron;
7. a sigmoid activation.

(a) Implement the constructor `__init__` to initialize all the layers needed for the model architecture.

Hint. *Calculate the number of input neurons of the first linear layer by evaluating the dimension of the output of the max-pooling layer, knowing that input images are subsampled so they have shape* $224 \times 224 \times 3$.

(b) Implement the method `forward` which takes in a batch of data `x` and performs the forward pass.

Training a Neural Network in PyTorch

We now need to train this network. We use the binary cross-entropy loss, defined as

$$L_{\text{BCE}}(w) = -\frac{1}{N}\sum_{n=1}^{N} \mathbf{y}_n \ln[h(\mathbf{x}_n, w)] + (1 - \mathbf{y}_n)\ln[1 - h(\mathbf{x}_n, w)], \tag{8}$$

where $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)$ are data points and h is the neural network. Here, $\mathbf{y}_n \in \{0, 1\}$ and $\mathbf{y}_n = 0$ is interpreted as the part being defective.

PyTorch has already implemented a variety of common losses and optimizers to ease training. In particular, this loss is available as `torch.nn.BCELoss`, and we use the optimizer `torch.optim.Adam` [5] to optimize it. The standard PyTorch training loop consists of the following steps:

- Collect a new batch of training data;
- Reinitialize the gradients of the network by calling the `zero_grad` method of the optimizer;
- Perform the forward pass by evaluating the model's prediction;
- Compute the resulting loss;
- Perform the backward pass by calling the `backward` method of the loss;
- Take a gradient step by calling the `step` method of the optimizer.

2.2. Fill in the training loop in the function `train`.

2.3. Train your network and evaluate it on the validation data set.

You will notice that the `SimpLeNet` does not achieve very good results on this task.

3 Fine-tuning Pre-trained Models in PyTorch

The example of the `SimpLeNet` hints at two general rules of thumbs when it comes to training neural networks: bigger networks are generally more performant but are also more expensive to train, both in terms of time and computational power. The size of the network directly correlates with the complexity of the learned model; wider networks have more basis functions, and deeper networks have more complex ones. A way to limit the overhead of training large networks is to use *pre-trained* ones. This is what we do in this section of the assignment. We propose to use a variant of the VGG architecture [6] for the pre-trained model and to specialize it for our binary classification task.

Using Pre-trained Networks There are two different rationales when using pre-trained networks, and they result in different training procedures. The first one is to use the pre-training as a better initialization of the weights. The second one is to set the weights of the pre-trained network as *non-trainable* and to replace its last layers with a custom trainable model, which is then called the *head*. The hope is that the non-trainable part extracts relevant features that are easier to work with for the small trainable head. This has the advantage of limiting the number of weights to train, thus easing training. This is the approach we adopt here: we remove some of the last fully-connected layers of the VGG network.

Trainable vs. Non-trainable Parameters PyTorch does not have the distinction between trainable and non-trainable parameters; it has only tensors. These tensors have an attribute `requires_grad` which indicates whether the gradients should be evaluated when performing back-propagation. Modules can set this attribute for all their parameters by calling the function `mod.requires_grad_(requires_grad)`, where `mod` is an instance of `nn.Module` and `requires_grad` is a boolean.

3.1. Implement the `forward` method of the class `AdaptedVGG`.

Remark. *Instantiating an `AdaptedVGG` for the first time will download the weights of VGG-11 from the PyTorch website (*507 MB*). Make sure you have a stable internet connection for this.*

(a) Fill in the missing part of the constructor to make sure that the gradients of the `vgg` module are not computed.

(b) Implement the method `forward` that performs a forward pass through the neural network.

Now that our network is more complex, it is also more prone to overfitting. The two simplest methods to fight overfitting in neural networks are *early stopping* and *dropout*. The first one is about stopping training after the validation loss has stopped decreasing for a sufficient number of steps. The second one consists in randomly hiding a fraction of the neurons of a layer during training (that is, setting their output to 0). Doing so, we hope to prevent the network to learn spurious correlations between features on the training set and, thus, to reduce overfitting [7].

3.2. Implement the function `early_stopping_check`. It takes in a history of losses `losses` and an integer `threshold` that defines after how many epochs of monotonically increasing losses the training should be stopped. It returns a boolean value indicating whether the training should stop.

Remark. *This is not the only possible criterion for early stopping. For instance, an alternative is to stop training if the average loss increases over the last `threshold` epochs.*

3.3. Implement the class `DropoutAdaptedVGG`. It has the same architecture as `AdaptedVGG` with an additional dropout layer between VGG and the custom head. Use `p=0.2` when creating the dropout layer.

Hint. *Dropout is implemented in Pytorch as `nn.Dropout`.*

Since training networks can take a long time, a good practice is to store checkpoints regularly, e.g., after each training epoch.

3.4. Implement the function `store_checkpoint` which takes in a `model` and the current `epoch` of training and stores the model as a checkpoint `.pt` file to the `checkpoints/` folder in a file named `model-[EPOCH].pt`, where `[EPOCH]` is the epoch number.

3.5. Fill in the training loop in the function `transfer_train` and train your model.

Remark. *You will notice the lines* `model.train()` *and* `model.eval()` *at the beginning of the training loop and before evaluating the model on the validation data set. They respectively put the model in training and evaluation mode. Some layers behave differently depending on this mode. For instance, dropout layers are deactivated at evaluation: it does not make sense to randomly hide some neurons when performing inference. VGG itself has dropout layers; this is the reason for the calls to* `self.vgg.eval()` *in the constructor of* `AdaptedVGG` *and the redefinition of its* `train` *method. Indeed, we want to make sure that the* `vgg` *module is always in* `eval` *mode so that its internal dropout layers are inactive.*

We can now test the performance of our model on the test set.

3.6. *(Optional)* Fill in the function `evaluate_accuracy` to compute the accuracy of the trained model on the test dataset, and evaluate it on your trained model.

References

[1] M. Plotke. “Kernel (image processing).” (2013), [Online]. Available: https://commons.wikimedia.org/wiki/File:Vd-Orig.png.

[2] N. Kantesaria, P. Vaghasia, J. Hirpara, and R. Bhoraniya. “Casting product image data for quality inspection.” (2020), [Online]. Available: https://www.kaggle.com/ravirajsinh45/real-life-industrial-dataset-of-casting-product.

[3] M. AI. “A gentle introduction to torch.autograd.” (), [Online]. Available: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html.

[4] M. AI. “Torch.nn.” (), [Online]. Available: https://pytorch.org/docs/stable/nn.html.

[5] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[6] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *arXiv preprint arXiv:1207.0580*, 2012.