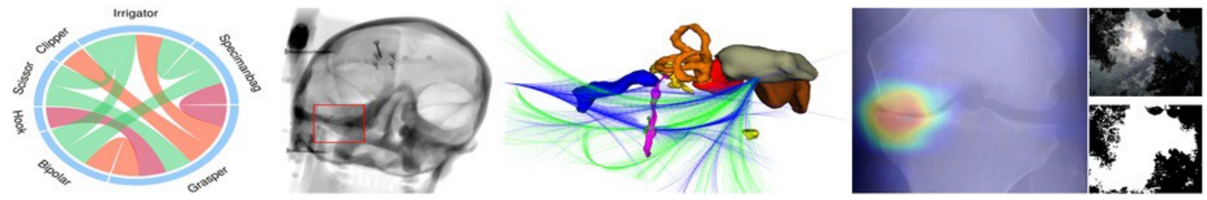
Only you can see this message

This story's distribution setting is on. Learn more



Project Roadmap for the Medical Imaging Student working with Deep Learning

Camila Gonzalez Follow
Aug 23 · 14 min read

As a student tackling a new Deep Learning project in the field of Medical Imaging, or even Machine Learning in general, you may be baffled about where to start. I've personally wasted a lot of time (on different occasions) by starting out testing different ideas and models, only to later forget about a lot of what I had done. I hope the advice in this post helps you avoid the basic mistakes I made and build up a code base that improves over time.

. . .

Setting up your project

Before you start delving with your data and models, it is useful to set up your project in an organized way from the start. For this:

- Adhere to a good package structure that allows other people to install all dependencies and run your code without needing much time. Including *requirements.txt* and *setup.py* files in Python projects can help a lot.
- As far as possible, make your code compatible with different operating systems. Even if you decide not to make your code public, your supervisor should be able to run it. For instance, avoid manipulating paths manually.
- Choose a testing library and integrate it into your package structure. Many people who work on Machine Learning avoid writing tests because training runs are difficult to reproduce. However, even if you only write tests for certain parts of your code, it will let you make changes more confidently and help other people understand your code better. *Pytest* is a very uncluttered library for Python. You can declare tests by prefixing functions with *test_* and run all tests by executing *pytest* from the root directory.

```
def test_f():
  assert True
```

- If you use a git server (e.g. *GitHub*), make sure your *.gitignore* is adapted to what you want to store in your repository. You can base your file in a template and adapt it to your needs. You'll likely exclude the metadata from your editor or IDE, as well as cached data. However, you may or may not choose to store intermediate results like model checkpoints and result files. If you exclude these because of size concerns, back them up somewhere else. Also, make sure you are not uploading prepossessed data. Particularly if you are working with medical data, ask your supervisor about the best practices for storing it. Ignore directories and their content by including:

```
/ignored_directory/*
```

- Think from the start about what design you want your code base to take. Even if you change it further along, you will likely not change the entire structure. I like to separate my modules into: a) data retrieval, including an abstraction for building validation and test sets (if not already defined) and/or cross-validation folds, b) model definition (so that different architectures work interchangeable) c) model training d) evaluation, e) visualization, f) utilities with miscellaneous helper functions, and g) a "main" module or pipeline which may take the form of a Jupyter notebook. Make sure when using the IPython console to autoreload imported modules.

```
# Notebook
%load_ext autoreload
```

```
%autoreload 2

# IPhyton console, e.g. in VSCode
from IPython import get_ipython
get_ipython().magic('load_ext autoreload')
get_ipython().magic('autoreload 2')
```

- Try to get as much reproducibility as possible. How to do this and to what extent it can be ensured depends on the framework. For instance, getting totally reproducible results in *Keras* and *PyTorch* is only guaranteed under certain circumstances. Also use a seed for numpy random operations, for instance when dividing the dataset.

```
# Numpy
import numpy as np
np.random.seed(0)

# Keras with Tensorflow backend
import tensorflow as tf
import random as rn
rn.seed(1)
# Force single-thread use
session_conf =
tf.ConfigProto(intra_op_parallelism_threads=1,

inter_op_parallelism_threads=1)
from keras import backend as K
tf.set_random_seed(1234)
sess = tf.Session(graph=tf.get_default_graph(),
config=session_conf)
K.set_session(sess)

# PyTorch
import torch
torch.manual_seed(0)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

- Write functions to save and restore checkpoints. You may only have access to GPUs for a limited period of time, so setting this up is important. Make sure that when you stop and resume your training you get the same results than when you train uninterruptedly. This may require you to save the optimizer state as well as the model state, as the behavior of many optimizers changes depending on the epoch (for instance, learning rate decay reduces the learning rate further along the training). Also, take into account that model states can get quite large, so make sure you have enough disk space for all the states you want to save in an experiment.

```
# Keras: save and restore whole model + train configuration
from keras.models import load_model
model.save('<path>.h5')
model = load_model('<path>.h5')

# Keras: save and restore only parameter state
model.save_weights('<path>.h5')
model.load_weights('<path>.h5')

# PyTorch: save and restore model state
torch.save(model.state_dict(), '<path>.pth')
model.load_state_dict(torch.load('<path>.pth'))

# PyTorch: save and restore optimizer state
torch.save(optimizer.state_dict(), '<path>.pth')
optimizer.load_state_dict(torch.load('<path>.pth'))
```

- Write a manual calculation for each evaluation measure you use and regularly compare them with the measures calculated by your framework of choice. Also, be clear about whether you are calculating a measure on a batch or the whole training or validation datasets. Only save results calculated on the whole sets so that you are not confused later on about divergence between different experiments.

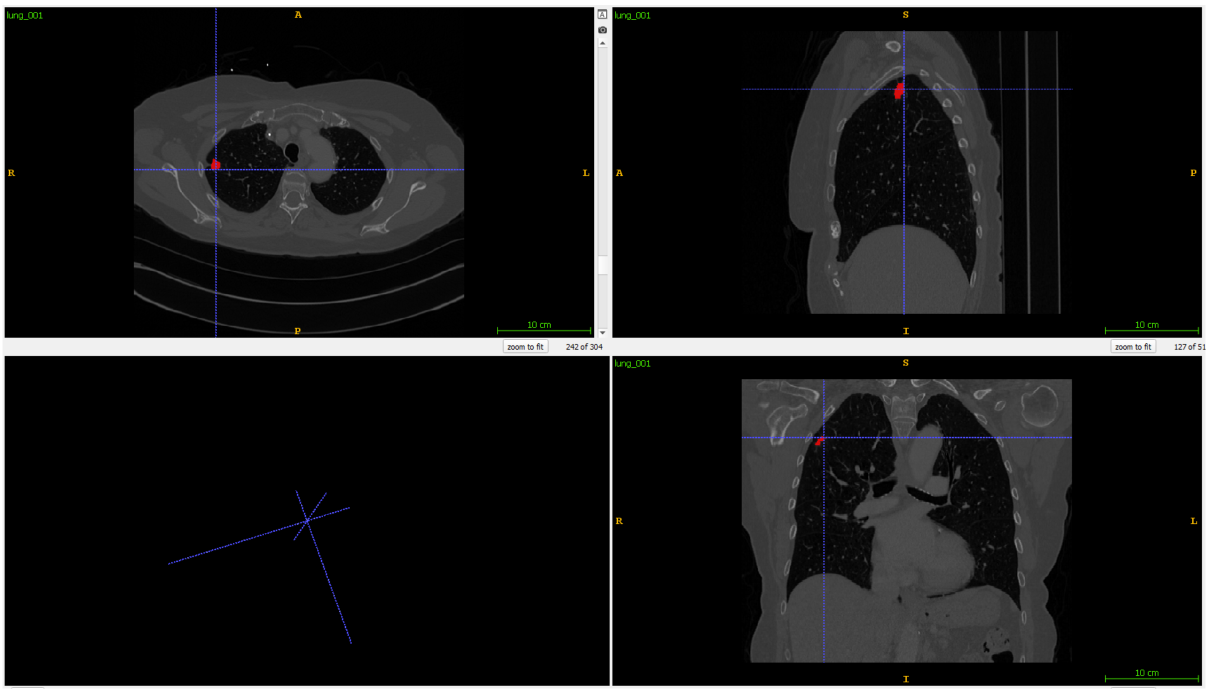
- Write functions to visualize each loss and performance measure. You may also want to visualize the distribution of the activations. Ideally, the mean of the activations should be zero and the variance should stay the same across every layer (ReLU layers typically reduce the variance).

. . .

Finding and inspecting data

- If you are unsure what dataset you want to work on and are interested in medical imaging, take a look at these lists of Medical Imaging datasets (1, 2, 3). Also explore Grand Challenges. You will usually get access to the data once you register for the challenge.
- The data will likely be in a medical data format, such as *.dicom*, *.nii*, etc. Choose an appropriate library to import it. For Python, this could be *pydicom* or *MedPy*.
- Inspect the class distribution and distribution of the data w.r.t. different aspects. For instance, for segmentation tasks calculate the distribution of the mask sizes for each class.
- Find and remove duplicates. Definitely make sure to do this before dividing the dataset.

- Inspect the data manually. The *ITK-SNAP* Toolkit is a good open-source choice to observe multi-dimensional medical data. See if you can yourself understand why the annotators have made certain decisions. If there are different annotators for a data sample, visualize these for the images they disagree on and see what the differences are. Also observe whether the decision may be based on texture or spatial position. In the first case, you may be able to work only on small patches, and in the second you may be able to down-sample the images to a greater extent.



CT segmentation visualized with the ITK-SNAP tool. The image is part of the Medical Segmentation Decathlon Lung Tumours dataset.

- If there are different annotators for a data sample, decide how to evaluate your results against their ground truths based on your manual inspection. For instance, for segmentation some doctors may annotate smaller areas than others. In this case, you may decide to use the set of annotations which more closely match what you believe is the area of interest. Find data points for which the disagreement is large so that you can later inspect your model's predictions manually for these examples.

- Finally, find a *dummy* dataset which resembles your dataset in terms of problem formulation, size and class distribution, but for which there are a lot more published results and which it is a lot faster to train on (because the image size is smaller and/or because it's a simpler problem so the training converges faster). This may be as simple as an adapted version of MNIST. Working on such a dataset alongside your chosen one will allow you to try out different things much faster, as well as to help identify errors if your performance falls to an unusual value for the simple dataset.

. . .

Designing a strong model

Once your code base is set up and your data is inspected, divided and ready to import, you can start working on designing your model. The goal of your project may be to obtain a good performance on some dataset or to explore a new method in some area (e.g. adversarial training, Auto-ML, interpretability). In the first case, following these steps may help you find your model. In the second, you will still need a good model which uses current techniques to validate your method, though you may prefer to keep it simple by avoiding using ensembles and data augmentation.

I like to roughly follow Andrej Karpathy's recipe, which consists on: 1. overfitting, 2. regularizing, 3. fine-Tuning. Below, I describe how I go about each step.

1. Overfitting

Include the following baselines:

- Simple models, such as 2D or 3D U-Nets for segmentation
- A state-of-the-art model which performs very well on similar datasets

Remove all regularization in the models and grow them until they have a near-perfect performance on the training data. Then, down-sample the images and/or crop them into small patches for as long as the training accuracy does not drop. If you are working on 3D data, see whether 2D models manage to capture the same information. Remember that the larger the data, the more GPU resources you will need.

Some things to keep in mind when training your baselines are:

Initialization: Use *He* initialization for ReLU (and variants) layers and *Xavier/Glorot* initialization for Tanh (particularly) and Sigmoid layers. These select the weights from a normal distribution with $\mu=0$ but a variance inversely proportional to the number of inputs. This *deeplearning.ai* post gives a good explanation about why these are good choices.

Optimizer and learning rate: As a starting point, test the performance of the Adam optimizer with its default learning rate, as well as a smaller one (e.g. 4e-5). Compare the results of different optimizers, including the simple SGD. Popular deep learning libraries include an array of optimizer implementations. First tune the learning rate. One way to do this is Leslie N. Smith's [1] technique (nicely explained on this blog post):

1. Start from a low learning rate and increase it exponentially for every batch, until the loss increases significantly (k times) above the best recorded loss
2. Plot the loss w.r.t. the learning rate (scaled logarithmically)
3. Calculate the point where the loss decreases the fastest (and check that this is consistent with the graph)

Afterwards, tune the decay rate and test out different decay functions, such as exponential decay or cosine decay.

Batch size: For the batch size, some recommend to select the maximum batch size that fits the GPU while others maintain that a smaller batch size improves the stability of batch normalization. Start with a greater batch size and try reducing it. When reducing the batch size, reduce the learning rate proportionally.

2. Regularizing

Some things you can try out to see if they improve the performance on the validation data are:

- **Dropout [2]:** Include dropout after dense or pooling layers. A good starting parameter is a 0.5 dropout rate which is later decreased (be aware that in some implementations, the parameter gives the probability NOT to drop out). For convolutional layers, regular dropout is not effective because neighboring pixels are highly correlated. Instead try out *spatial dropout* [3], which randomly sets entire feature maps (channels) to 0, before pooling layers.

```
# Keras
from keras.layers import Dropout, SpatialDropout2D
model.add(Dropout(0.5))
model.add(SpatialDropout2D(0.05, data_format=None))

# PyTorch
import torch.nn as nn
def __init__(self):
    self.dropout = nn.Dropout(0.5)
    self.spatial_dropout = nn.Dropout2d(0.05)
```

- **Gaussian Noise [4]:** Apply Gaussian noise to the input, or to the output of dense or pooling layers. Check out this useful post about adding noise to neural networks.

```
# Keras
from keras.layers import GaussianNoise
model.add(GaussianNoise(0.1))
```

```
# PyTorch: see https://discuss.pytorch.org/t/where-is-the-
noise-layer-in-pytorch/2887/4
```

- **Batch Normalization [5]:** Include batch normalization, particularly between convolutional layers, to make the input of every layer have approximately the same distribution in every batch. Place after Sigmoid and TanH activation functions and before for functions that may result in non-Gaussian distributions like ReLU.

```
# Keras
from keras.layers.normalization import BatchNormalization
model.add(BatchNormalization())

# PyTorch
import torch.nn as nn
def __init__(self):
    self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
    self.conv2_bn = nn.BatchNorm2d(20)
```

- **L1 and L2 Losses:** Include L1 and/or L2 penalties on the activation values. As a rule of thumb, the λ can be chosen to be close to the learning rate.

```
# Keras: added to layers
from keras import regularizers
model.add(Dense(64, input_dim=64,
                activity_regularizer=regularizers.l1(0.01)))

# PyTorch: added to the loss
linear1_params = torch.cat([x.view(-1) for x in
model.linear1.parameters()])
```

```
l1 = torch.norm(linear1_params, 1)
l2 = torch.norm(linear1_params, 2)
loss = cross_entropy_loss + lambda1 * l1 + lambda2 * l2

# PyTorch: as weight decay, see:
https://towardsdatascience.com/different-types-of-
regularization-on-neuronal-network-with-pytorch-a9d6faf4793e
```

- **Weight Decay [6]:** Prevent weights from growing too large.

```
# Keras: added to layers
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l1(0.01)))

# PyTorch: passed to optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.001,
weight_decay=1e-4)
```

- **Gradient Clipping [7]:** Clip gradients if their norms exceed some threshold to prevent exploding gradients. I have not found a good way to find an initial threshold, but visualizing the gradients may help find an appropriate value.

```
# Keras
from keras import optimizers
# Clip gradients to a max norm
sgd = optimizers.SGD(lr=0.01, clipnorm=1.)
# Clip gradients to be between 0.5 and -0.5
sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)

# PyTorch: clip gradients to a max norm
from torch.nn.utils import clip_grad_norm
clip_grad_norm_(model.parameters(), 1.)
```

- **Data Augmentation:** Include basic and then more creative ways to augment data, such as using GANs. Use existing implementations, such as the ImageDataGenerator for Keras, and transforms (apart from the ones you use for preprocessing) for PyTorch.
- **Pretraining:** Use a pretrained model as initialization. You can freeze the first layers to speed up training. Keep in mind that if the model was trained by someone else, the data may have to be normalized differently than between 0 and 1, so normalize it accordingly.

```
# Keras
from keras.applications import VGG16
vgg = VGG16(weights='imagenet', include_top=False,
input_shape=(image_size, image_size, 3))
# Freeze some layars
for layer in vgg.layers[:-4]:
    layer.trainable = False
# Start a new model and add the pretrained layers
model = Sequential()
model.add(vgg_conv)
# Add final layers
model.add(Flatten())
model.add(Dense(nr_outputs, activation='softmax'))

# PyTorch
import torchvision.models as models
model =
models.inception_v3(pretrained=model_config['pretrained'])
# Freeze some layars
for params in model.parameters()[:-4]:
    params.requires_grad = False
# Replace final layer
model.fc = nn.Linear(model.fc.in_features, nr_outputs)
```

- **Skip Connections [8]:** Include skip connections/residual blocks in the model to connect non-sequential layers. If using Keras, use the *functional API* and the *Model* class instead of *Sequential*. For PyTorch, implement skip connections in the model's *forward(self, x)* method.
- **Early Stopping:** The model may start overfitting once the validation loss starts increasing, so you can stop training there. However, the validation loss also fluctuates, so saving the model and comparing it to a later one may be preferable.

Add every modification separately and test it throughout to ensure is causes a performance boost.

3. Fine-tuning

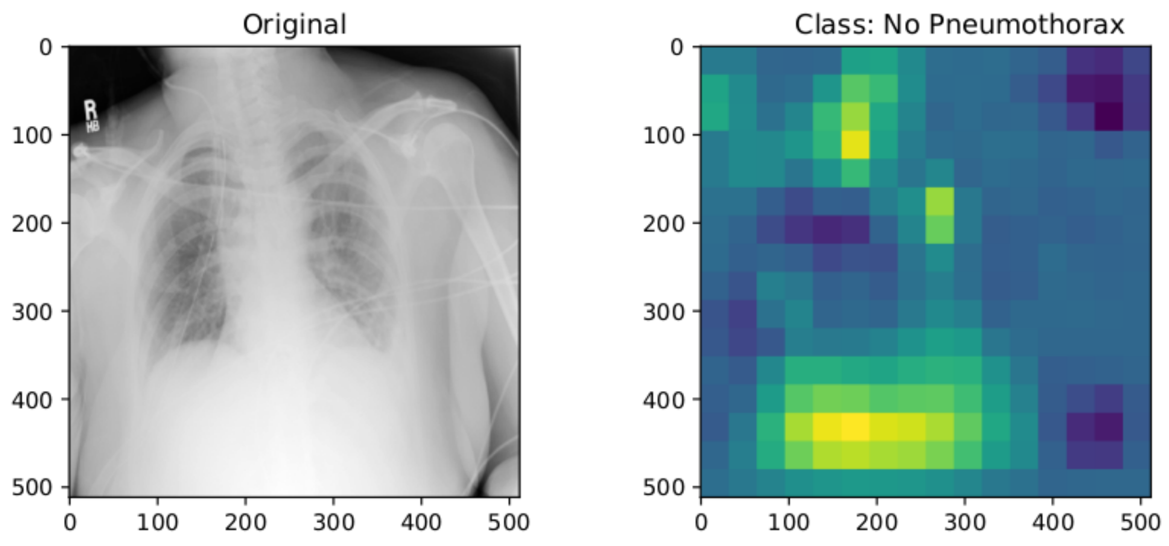
If you are focused on performance, some steps that may help you are:

- Tuning hyper parameters with random or grid search.
- Using ensembles of different models. The *nnU-Net* [9], which won the Medical Segmentation Decathlon last year, is a clear example that combining simple models may get you the best performance.

. . .

Interpreting your results

Increasingly, and particularly in the medical field, it is no longer enough to report a good performance of a model w.r.t. some metric. Instead, the model also has to be *interpretable*. There is no one accepted definition of interpretability that the Machine Learning community agrees on, but there are many techniques to gain insight into the model's hidden features and predictions, particularly for image data.



A class activation map for a chest x-ray image, showing which areas indicate the patient does not have pneumothorax desease. The data can be downloaded from Kaggle.

Many approaches work on a *post-hoc* manner without requiring changing the model. Some of these are:

- **Occluding parts of an image:** if you know what area of the image the network should be looking at for making a prediction, e.g. if you have trained a classier to predict a disease in an organ you have the segmentation for, you can test how your network performs if you cover that area.
- **Saliency Maps [10]:** to figure out how relevant a pixel is for a

class, the gradient of the output for the class with respect to the pixel is calculated. A greater gradient signifies that the pixel changing has a greater effect on the score for that class.

- **GradCAM [11]:** For an image and a class, a heatmap can be generated which indicates what areas of an image have caused an increased score for that class. For this, the gradient is calculated of the class of interest with respect to the feature maps of the last convolutional layer. Then, global-average-pooling is applied to get an importance value for each neuron. Multiplying these with the feature maps gives a matrix of values of the size of the feature maps. A ReLU activation is applied then applied so that only positive values remain. Rescaling the result produces the heatmap.
- **TCAV [12]:** Concept Activation Vectors can be used to test whether a concept (e.g. *stripes*) is used by a group of parameters to classify images as belonging to a class (e.g. *zebra*). For this, features are extracted for images representing that concept and for random counterexamples. A linear classifier is trained to divide both sets of features, and the direction perpendicular to the decision boundary is taken to test how relevant the concept is for a set of examples belonging to that class.

Depending on the framework you use, you may find openly available code for these techniques. Testing them on your model will help you understand why the model makes the decision it makes, and why it may not work for some examples.

. . .

Final thoughts

Last but not least, make sure you document your experiments. Write a short description of what you're working on as often as you can, and annotate the date on each model state and result file so you can find them later on. You may even want to save all the Jupyter notebooks you have run experiments on.

. . .

References

[1] Smith, L. N. (2017, March). Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)* (pp. 464–472). IEEE.

[2] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929–1958.

[3] Tompson, J., Goroshin, R., Jain, A., LeCun, Y., & Bregler, C. (2015). Efficient object localization using convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 648–656).

[4] Holmstrom, L., & Koistinen, P. (1992). Using additive noise in

back-propagation training. *IEEE transactions on neural networks*, *3*(1), 24–38.

[5] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[6] Krogh, A., & Hertz, J. A. (1992). A simple weight decay can improve generalization. In *Advances in neural information processing systems* (pp. 950–957).

[7] Pascanu, R., Mikolov, T., & Bengio, Y. (2013, February). On the difficulty of training recurrent neural networks. In *International conference on machine learning* (pp. 1310–1318).

[8] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

[9] Isensee, F., Petersen, J., Klein, A., Zimmerer, D., Jaeger, P. F., Kohl, S., … & Maier-Hein, K. H. (2018). nnu-net: Self-adapting framework for u-net-based medical image segmentation. *arXiv preprint arXiv:1809.10486*.

[10] Simonyan, K., Vedaldi, A., & Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*.

[11] Kim, B., Wattenberg, M., Gilmer, J., Cai, C., Wexler, J., Viegas, F., & Sayres, R. (2017). Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav). *arXiv preprint arXiv:1711.11279*.

[12] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 618–626).

Machine Learning Medical Imaging Deep Learning Visual Computing