Practice of Deep Learning

Bonus Reading 2 – Good Practices

Matthieu De Mari

About this lecture

1. What is the **train-test-validation split**?
   Why is it **good practice** to use an extra validation set?
2. What is the **early stopping** of optimizer concept?
   Why is it **good practice** to use it?
3. What are **saver** and **loader** functions?
   Why is it **good practice** to use them?
4. What are **other common good practices** when it comes to Neural Networks?
5. How to decide on **an appropriate number of layers and neurons**?

Good Practice #1: Using additional performance metrics beyond loss

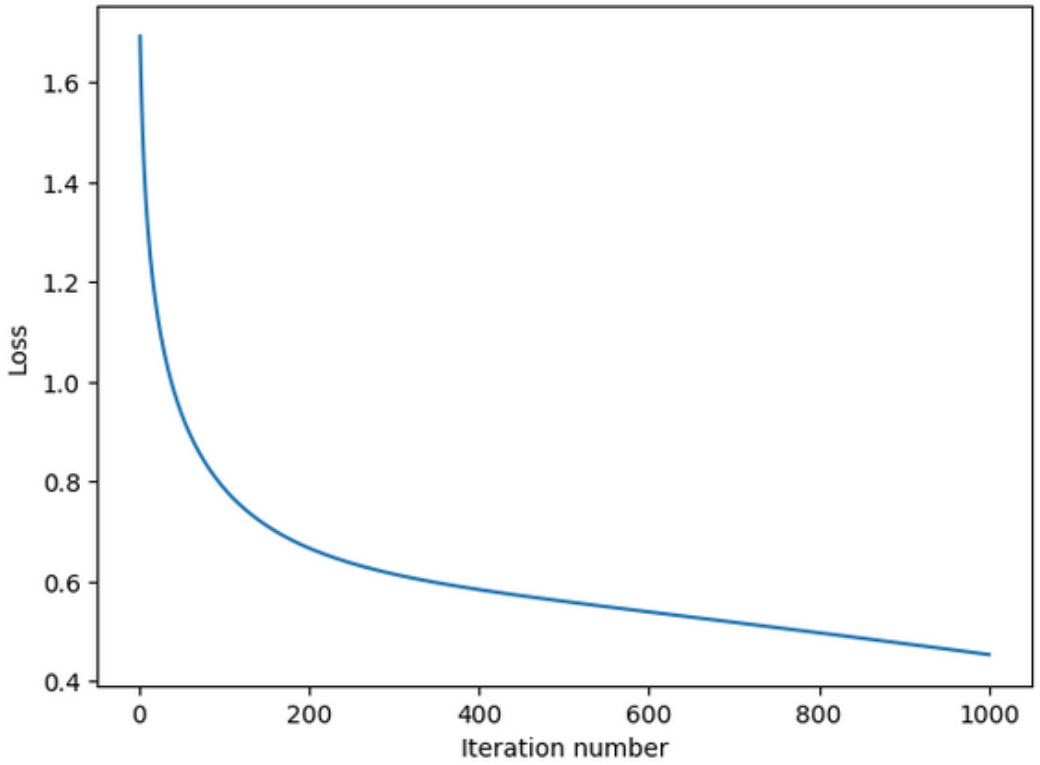
Good practice #1: Using additional performance metrics beyond the loss function used for training.

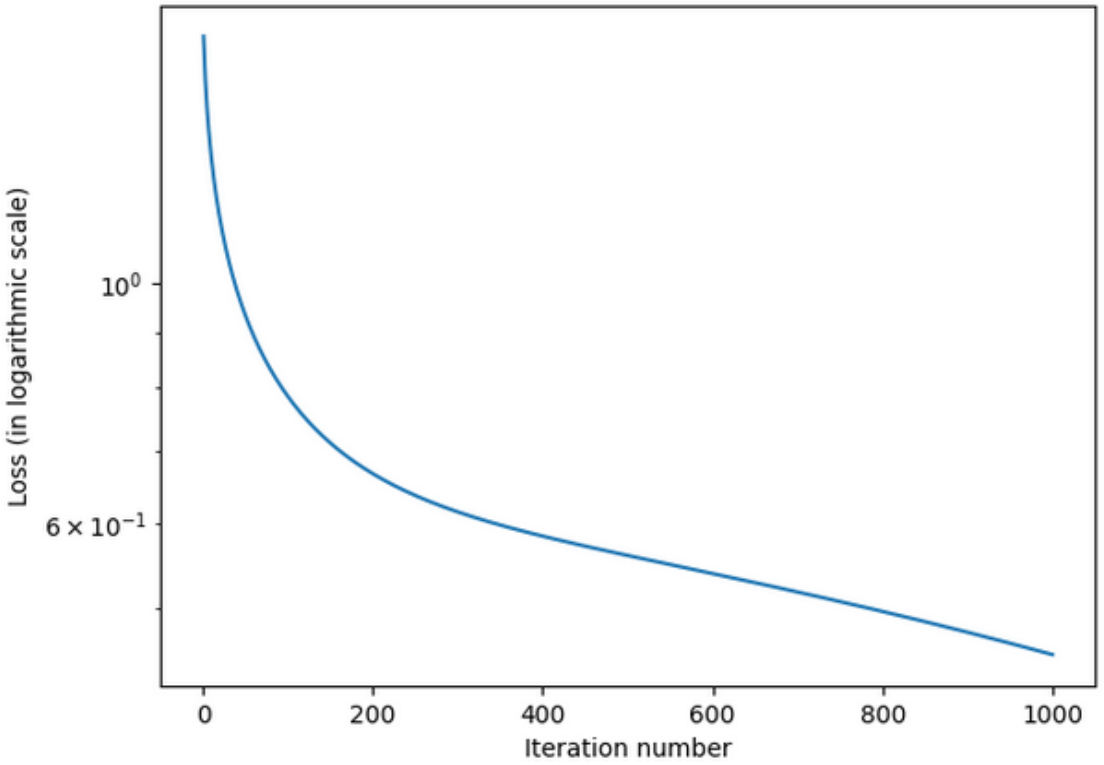
- Loss functions are great functions to minimize during training (for instance MSE is convex, which is a nice property to have).
- But these loss functions do not always carry a meaning that is easily interpretable for humans. **A low loss does not mean that a model has successfully trained and learnt to perform the task!**
- **For instance, the cross-entropy loss function we have used in binary classification does not tell us much about the actual performance of the model. Is it accurate? Or not?**
- Recall notebook 4, when we had no activation functions.

Good Practice #1: Using additional performance metrics beyond loss

Counter-example from Notebook 4:

We were able to train, if we consider the loss evolution curves…



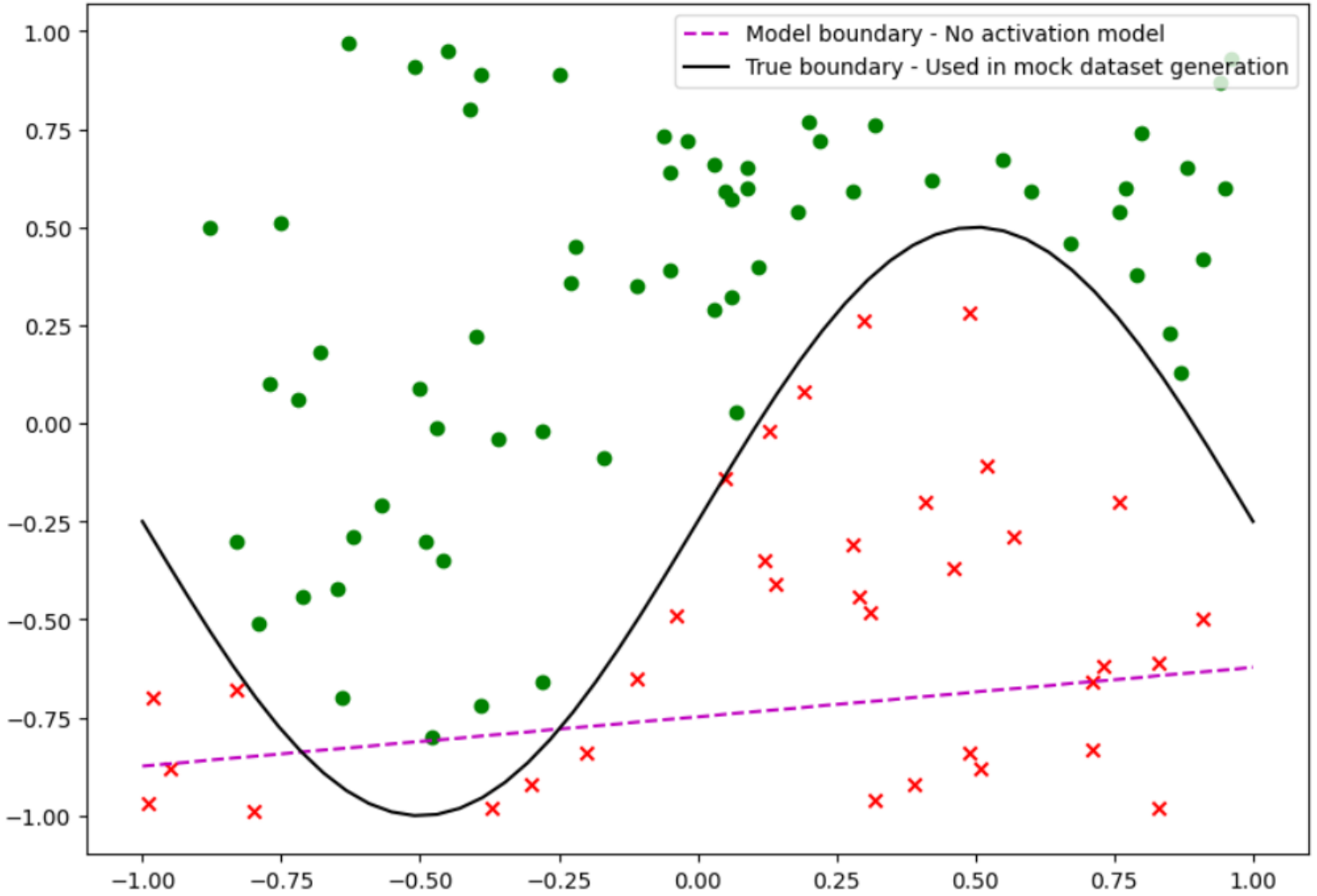


Good Practice #1: Using additional performance metrics beyond loss

Counter-example from Notebook 4:

If we consider the **accuracy** of the model, however…

Then it is fairly obvious that the model failed to produce the correct classification.



Good Practice #1: Using additional performance metrics beyond loss

As such, it would be a good idea to have **some more interpretable performance metrics on the side**, such as the accuracy for instance, to display along with the loss function during training.

This could confirm that the model succeeds at the task (or not).

1. Let us start by adding an accuracy method to our model.

```
43      def accuracy(self, inputs, outputs):
44          # Calculate accuracy for given inputs and ouputs
45          pred = [int(val >= 0.5) for val in self.forward(inputs)]
46          acc = sum([int(val1 == val2[0]) for val1, val2 in zip(pred, outputs)])/outputs.shape[0]
47          return acc
```

Good Practice #1: Using additional performance metrics beyond loss

As such, it would be a good idea to have **some more interpretable performance metrics on the side**, such as the accuracy for instance, to display along with the loss function during training.

This could confirm that the model succeeds at the task (or not).

2. We can then add another list for tracking accuracies, along with losses during training.

```
104     def train(self, inputs, outputs, N_max = 1000, alpha = 1e-5, beta1 = 0.9, beta2 = 0.999, \
105               delta = 1e-5, batch_size = 100, display = True):
106         # Get number of samples
107         M = inputs.shape[0]
108 
109         # List of losses, starts with the current loss
110         self.losses_list = [self.CE_loss(inputs, outputs)]
111         self.accuracies_list = [self.accuracy(inputs, outputs)]
```

Good Practice #1: Using additional performance metrics beyond loss

As such, it would be a good idea to have **some more interpretable performance metrics on the side**, such as the accuracy for instance, to display along with the loss function during training.

This could confirm that the model succeeds at the task (or not).

3. During training we will update the accuracy of the model, appending to list, and displaying them during training along with loss.

```
134             # Update accuracies
135             acc = self.accuracy(inputs, outputs)
136             self.accuracies_list.append(acc)
137 
138             # Display
139             if(display and iteration_number % (N_max*0.05) == 1):
140                 print("Iteration {} - Loss = {} - Acc = {}".format(iteration_number, self.loss, acc))
```

Good Practice #1: Using additional performance metrics beyond loss

As such, it would be a good idea to have **some more interpretable performance metrics on the side**, such as the accuracy for instance, to display along with the loss function during training.

This could confirm that the model succeeds at the task (or not).

4. Finally, amend our performance curves to show evolution of accuracy over the training iterations.

```
164         axs[2].plot(list(range(len(self.accuracies_list))), self.accuracies_list)
165         axs[2].set_xlabel("Iteration number")
166         axs[2].set_ylabel("Accuracy")
167         # Display
168         plt.show()
```

Good Practice #1: Using additional performance metrics beyond loss

When the model is trained, we can see the evolution of the accuracy performance metric.

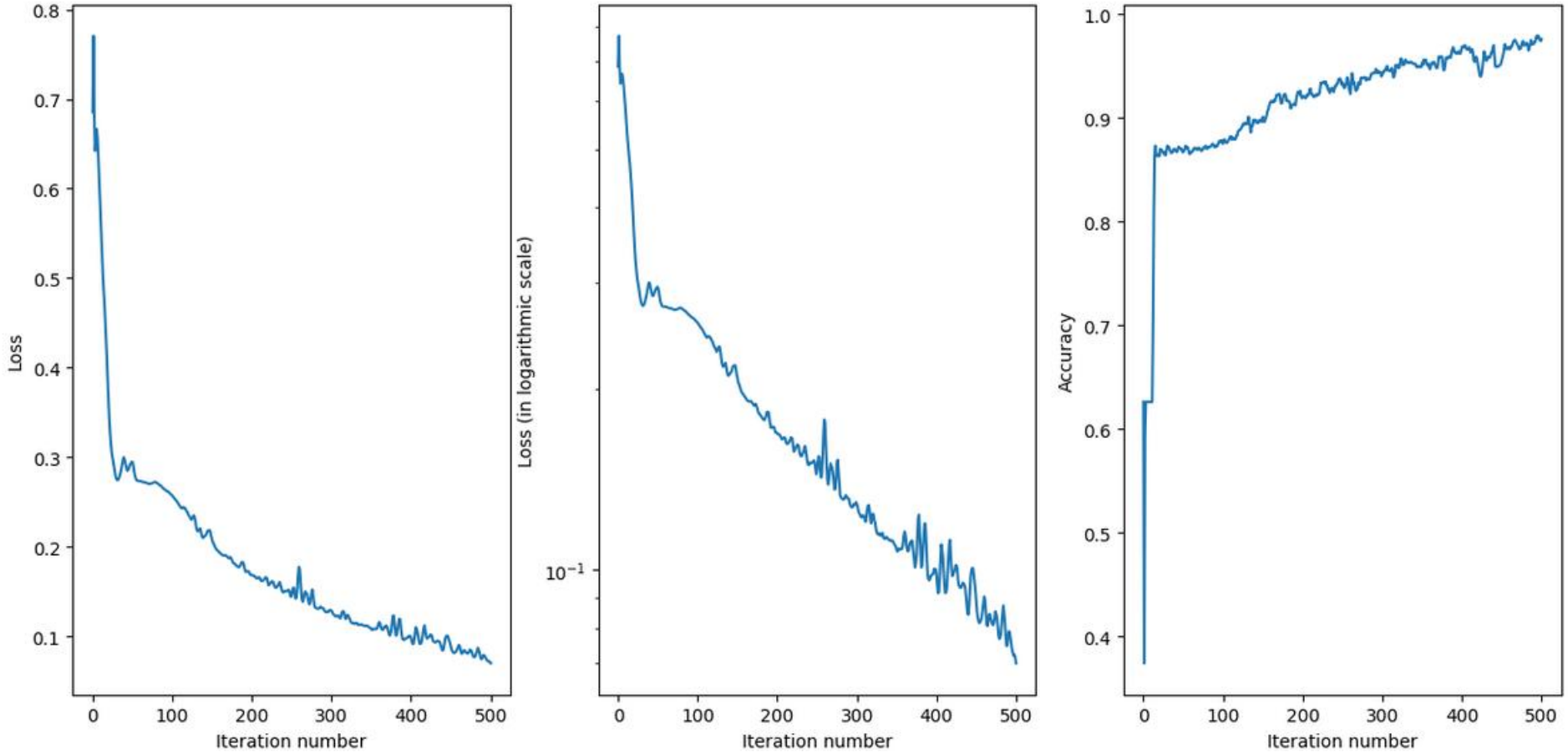
This accuracy score is a more interpretable metric for humans.

Here, it shows that the model accuracy increases and approaches 100%, which is a (very) good sign.

```
Iteration 1 - Loss = 0.7704676641856629 - Acc = 0.374
Iteration 26 - Loss = 0.29408041996849577 - Acc = 0.867
Iteration 51 - Loss = 0.2907713475859916 - Acc = 0.868
Iteration 76 - Loss = 0.2714162997128199 - Acc = 0.871
Iteration 101 - Loss = 0.25587504504663494 - Acc = 0.879
Iteration 126 - Loss = 0.23378145473572498 - Acc = 0.893
Iteration 151 - Loss = 0.2045784244297011 - Acc = 0.897
Iteration 176 - Loss = 0.18302469671030092 - Acc = 0.922
Iteration 201 - Loss = 0.167838515123934 - Acc = 0.921
Iteration 226 - Loss = 0.16116229703195345 - Acc = 0.934
Iteration 251 - Loss = 0.1512993470399631 - Acc = 0.931
Iteration 276 - Loss = 0.15216076879868856 - Acc = 0.938
Iteration 301 - Loss = 0.12705500255256222 - Acc = 0.944
Iteration 326 - Loss = 0.11463009619046385 - Acc = 0.951
Iteration 351 - Loss = 0.10719722692507941 - Acc = 0.951
Iteration 376 - Loss = 0.113238740132263 - Acc = 0.953
Iteration 401 - Loss = 0.09424265876829825 - Acc = 0.969
Iteration 426 - Loss = 0.10069262359991561 - Acc = 0.949
Iteration 451 - Loss = 0.08506155393228489 - Acc = 0.958
Iteration 476 - Loss = 0.08332931024313735 - Acc = 0.973
Stopping - Maximal number of iterations reached.
0.06991123688339018
```

Good Practice #1: Using additional performance metrics beyond loss

And of course, additional metrics means additional training curves.



Good Practice #2: Using a train-test-valid split

Good practice #2: Using a train-test-validation split.

If you allow the model to train, you will most likely realize that the accuracy reaches 100% (or close to 100% after a few iterations).

The loss could, however, be minimized even further.

This raises a few questions:

- Should we have stopped when the accuracy was already 100%?

- Are we at risk of overfitting by pursuing the training in an attempt to minimize the loss even more, even though it will not improve the accuracy further?

Good Practice #2: Using a train-test-valid split

While it is true that we are using the minimization of the loss function to train our model and decide on its parameters...

Our ultimate objective was NEVER to minimize the training loss function (or maximize the accuracy we obtain on training samples).

Our objective is and always has been **generalization**, that is training a model that will **generalize well to unseen data**.

This was the rationale for using a **train-test split** earlier.

Good Practice #2: Using a train-test-valid split

In Week 1, we introduced the concept of **generalization** and discussed how it is common practice **not to evaluate the performance of the model on the same samples that have been used for training**.

This led us to define two sets, called **train** and **test** sets, and we would:

- Use the **train** samples in the **backpropagation/training** procedure,
- And use the **test** samples to **objectively evaluate the model** and assess if it is able to **generalize** well or not.

We will push it one step further, by introducing the concept of **train, test and validation samples**.

Good Practice #2: Using a train-test-valid split

Good practice #2: Using a train-test-validation split.

During the data preparation procedure, we will now split the data in three sets:

- **Training** samples
- **Validation** samples
- **Test** samples.

The **training set** is used to train the machine learning model.

- The model is presented with many examples from the training set, and the model "learns" to make good predictions based on those examples.
- For the training procedure to be efficient, **most samples from the dataset should go into the training set**.

Good Practice #2: Using a train-test-valid split

After each iteration of training, the model is tested on the **validation set** to **assess the performance of the model** and see **how well it would generalize to unseen data**.

- We will use it to **spot underfitting and overfitting** in training.
- The **validation set** requires less samples than the **training set**.
- The **validation set** will also be used to **choose the best hyperparameters** for training the model (Good Practice #3-4).
- The **validation set** will also be used to decide on the best model parameters to use after several training rounds (Good Practice #3-4).

For this reason, it **can NOT be used to compute an objective performance metric** to measure the generalization capabilities of our model.

Good Practice #2: Using a train-test-valid split

As before, the **test set** is a set of data that **the model has NOT SEEN during the training or validation** process.

- It is used to **evaluate the final performance** of the model after training is complete, hence providing an estimate of **how well the model will perform on truly unseen data (generalization!)**.
- The **testing set** is usually set to have as many samples as the **validation set**.

Good Practice #2: Using a train-test-valid split

Question: What is considered a good repartition for the **train-test-validation** split?

The core idea is that:

1. **We want as many samples as possible in the training set,** as this helps training of the model.
2. **We want enough samples in validation/test sets,** to ensure diversity and that the **law of large numbers** applies**.**

Reminder from the past (Law of large numbers): When calculating a mean/average metric, like MSE, we need enough samples so that the empirical approximation matches closely the theoretical value.

Good Practice #2: Using a train-test-valid split

<u>Question:</u> What is considered a good repartition for the t**rain-test-validation** split?

Following the two ideas from earlier.

Scenario 1: you have more than ~10000 samples (large dataset).

Scenario 2: you have less than ~10000 samples (small dataset).

Scenario 1: you have more than ~10000 samples (large dataset).

- Put ~**1000 samples** in the **validation set**. And put another ~**1000 samples** in the **test set**.
- (Having 1000 samples should be enough for law of large numbers to apply).
- Put **the rest** in **training set**.

Good Practice #2: Using a train-test-valid split

<u>Question:</u> What is considered a good repartition for the **train-test-validation** split?

Following the two ideas from earlier.

Scenario 1: you have more than ~10000 samples (large dataset).

Scenario 2: you have less than ~10000 samples (small dataset).

Scenario 2: you have less than ~10000 samples (small dataset).

- Put ~**10% of samples** in the **validation set**. And put another ~**10%** in the **test set**.
- (We might not have enough samples to ensure the law of large numbers applies, but we try our best).
- Put **the remaining ~80%** in **training set**.

Good Practice #2: Using a train-test-valid split

Here, we will "mimic" a splitting by using our dataset generator three times, with different seeds and number of samples each time.

```python
# Generate dataset (train)
np.random.seed(47)
n_points_train = 1000
train_val1_list, train_val2_list, train_inputs, train_outputs = create_dataset(n_points_train, min_val, max_val)
print(train_inputs.shape)
print(train_outputs.shape)
```

```
(1000, 2)
(1000, 1)
```

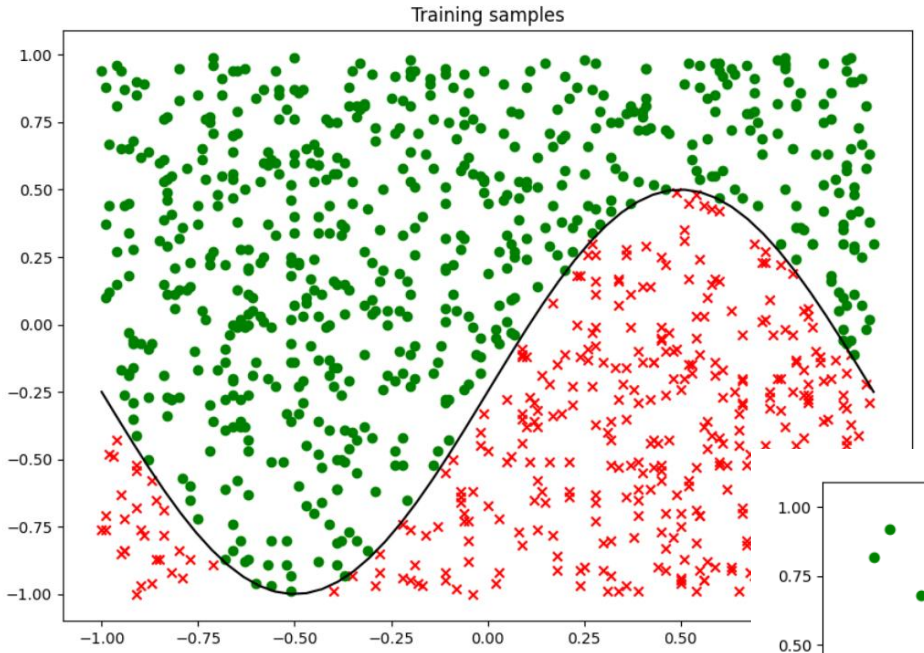
```python
# Generate dataset (test)
np.random.seed(17)
n_points_test = 100
test_val1_list, test_val2_list, test_inputs, test_outputs = create_dataset(n_points_test, min_val, max_val)
print(test_inputs.shape)
print(test_outputs.shape)
```

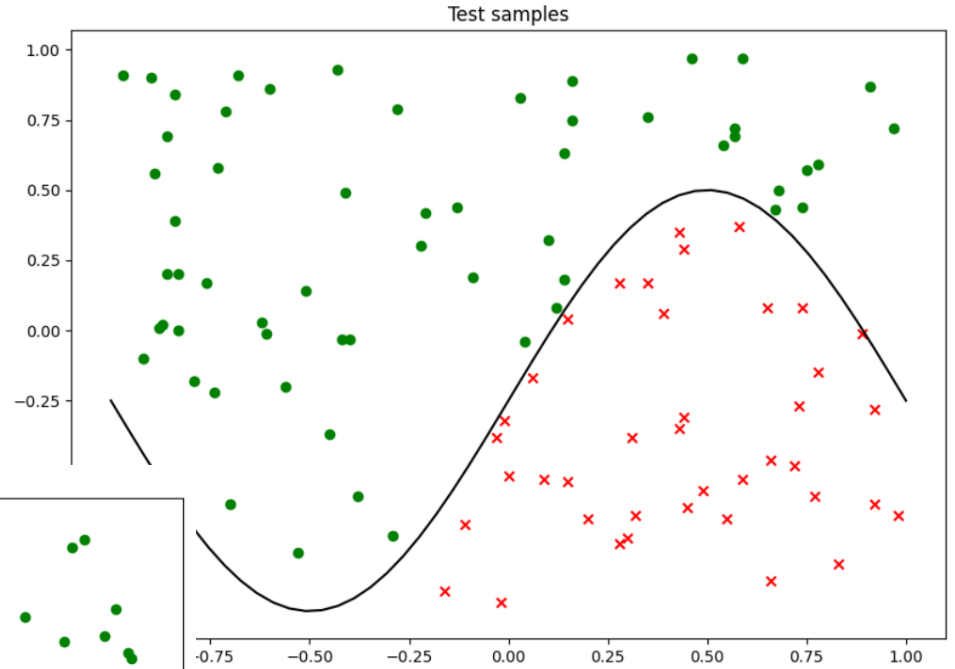
```
(100, 2)
(100, 1)
```

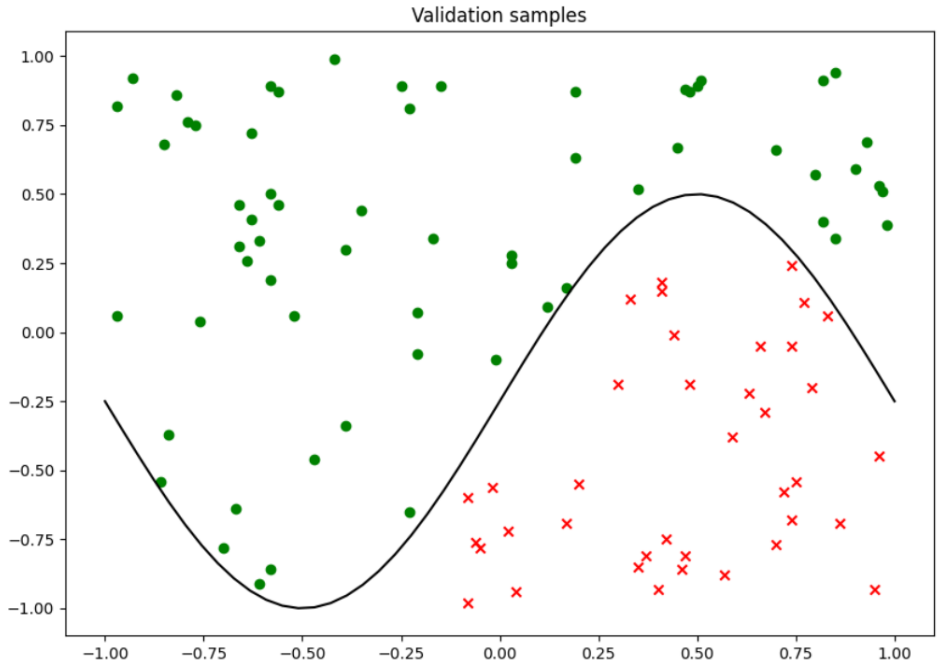
```python
# Generate dataset (valid)
np.random.seed(27)
n_points_valid = 100
valid_val1_list, valid_val2_list, valid_inputs, valid_outputs = create_dataset(n_points_valid, min_val, max_val)
# Check a few entries of the dataset
print(valid_inputs.shape)
print(valid_outputs.shape)
```

```
(100, 2)
(100, 1)
```

Good Practice #2: Using a train-test-valid split

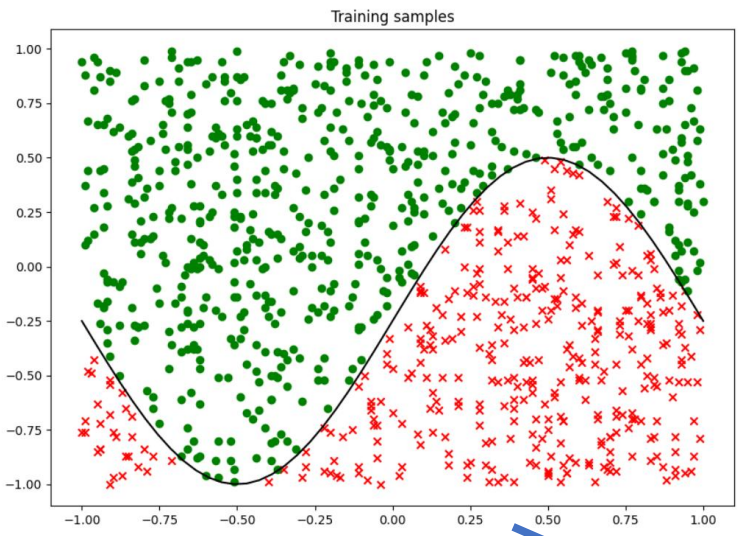


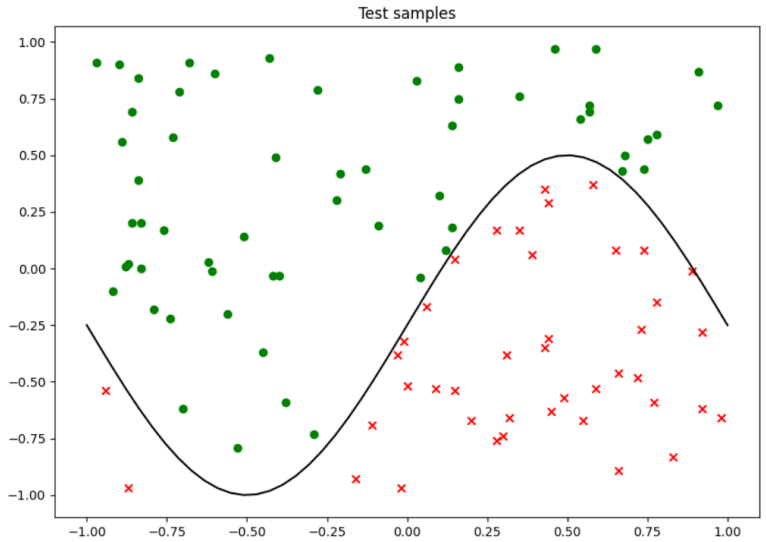


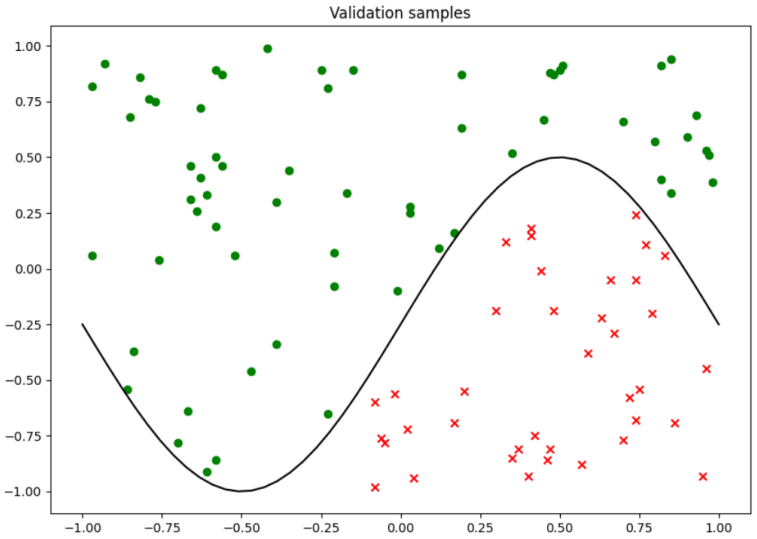


Good Practice #2: Using a train-test-valid split

During training pass all three datasets as parameters








```
104     def train(self, train_inputs, train_outputs, test_inputs, test_outputs, valid_inputs, valid_outputs, \
105               N_max = 1000, alpha = 1e-5, beta1 = 0.9, beta2 = 0.999, \
106               delta = 1e-5, batch_size = 100, display = True):
```

Good Practice #2: Using a train-test-valid split

A few more changes…

1. Track losses and accuracies during training on both training and validation sets.

```
110         # List of losses and accuracies
111         self.train_losses_list = [self.CE_loss(train_inputs, train_outputs)]
112         self.train_accuracies_list = [self.accuracy(train_inputs, train_outputs)]
113         self.valid_losses_list = [self.CE_loss(valid_inputs, valid_outputs)]
114         self.valid_accuracies_list = [self.accuracy(valid_inputs, valid_outputs)]
```

```
132                 # Update losses list
133                 train_loss = self.CE_loss(train_inputs, train_outputs)
134                 self.train_losses_list.append(train_loss)
135                 valid_loss = self.CE_loss(valid_inputs, valid_outputs)
136                 self.valid_losses_list.append(valid_loss)
137 
138                 # Update accuracies
139                 train_acc = self.accuracy(train_inputs, train_outputs)
140                 self.train_accuracies_list.append(train_acc)
141                 valid_acc = self.accuracy(valid_inputs, valid_outputs)
142                 self.valid_accuracies_list.append(valid_acc)
```

Good Practice #2: Using a train-test-valid split

A few more changes…

2. On each iteration, display losses and accuracies on both the training and validation sets.

```
# Display
if (display):
    message = "Iteration {} ".format(iteration_number)
    message += "\n - Train Loss = {} - Train Acc = {}".format(train_loss, train_acc)
    message += "\n - Validation Loss = {} - Validation Acc = {}".format(valid_loss, valid_acc)
    print(message)
```

Good Practice #2: Using a train-test-valid split

A few more changes…

3. After training has completed, check accuracy values (and, optionally, loss values) on the test set for a final confirmation, on the model's generalization capabilities.

```
# Display Accuracy on test
print("Test accuracy = {}".format(self.accuracy(test_inputs, test_outputs)))
```

Good Practice #2: Using a train-test-valid split

Training the model now gives the following full display.

In general, three possibilities:

- **Poor** values on both **train** metrics and **validation** metrics → **Underfitting**.
- **Good** values on **train** metrics, **poor** values on **validation** metrics → **Overfitting**.
- **Good** values on both **train** and **validation** metrics → **Great!**

```
Iteration 1
 - Train Loss = 0.6562206116565713 - Train Acc = 0.626
 - Validation Loss = 0.6592760519158617 - Validation Acc = 0.62
Iteration 31
 - Train Loss = 0.2830951890174539 - Train Acc = 0.869
 - Validation Loss = 0.20063836446117492 - Validation Acc = 0.91
Iteration 61
 - Train Loss = 0.2656823064976081 - Train Acc = 0.876
 - Validation Loss = 0.19536548122888026 - Validation Acc = 0.91
Iteration 91
 - Train Loss = 0.2051948666756941 - Train Acc = 0.91
 - Validation Loss = 0.12551601511035104 - Validation Acc = 0.98
Iteration 121
 - Train Loss = 0.15131533920216939 - Train Acc = 0.934
 - Validation Loss = 0.10093479006170472 - Validation Acc = 0.99
Iteration 151
 - Train Loss = 0.1115951833149475 - Train Acc = 0.955
 - Validation Loss = 0.07298127188796165 - Validation Acc = 0.99
Iteration 181
 - Train Loss = 0.07802364083046055 - Train Acc = 0.98
 - Validation Loss = 0.05391519243728255 - Validation Acc = 1.0
Iteration 211
 - Train Loss = 0.05500636101156745 - Train Acc = 0.991
 - Validation Loss = 0.039277569946238516 - Validation Acc = 1.0
Iteration 241
 - Train Loss = 0.04363952867730853 - Train Acc = 0.993
 - Validation Loss = 0.03176801602040892 - Validation Acc = 1.0
Iteration 271
 - Train Loss = 0.03726340017086438 - Train Acc = 0.993
 - Validation Loss = 0.02735269123594779 - Validation Acc = 1.0
Stopping - Maximal number of iterations reached.
Test accuracy = 1.0
```

Good Practice #2: Using a train-test-valid split

A few more changes…

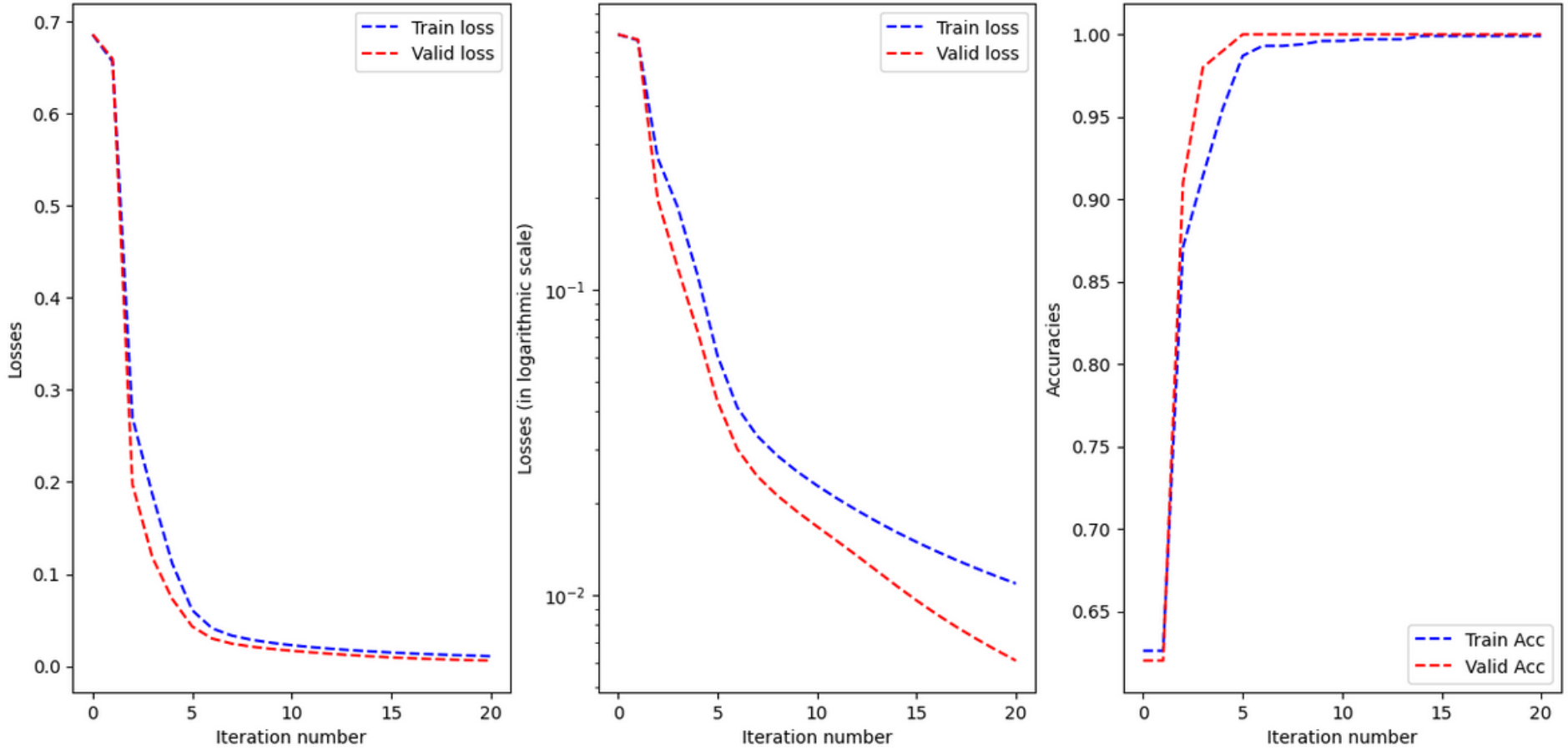
4. Training curves will now include training and validation values.

```
168     def show_training_curves(self):
169         # Initialize matplotlib
170         fig, axs = plt.subplots(1, 3, figsize = (15, 7))
171         axs[0].plot(list(range(len(self.train_losses_list))), \
172                     self.train_losses_list, "b--", \
173                     label = "Train loss")
174         axs[0].plot(list(range(len(self.valid_losses_list))), \
175                     self.valid_losses_list, "r--", \
176                     label = "Valid loss")
177         axs[0].set_xlabel("Iteration number")
178         axs[0].set_ylabel("Losses")
179         axs[1].plot(list(range(len(self.train_losses_list))), \
180                     self.train_losses_list, "b--", \
181                     label = "Train loss")
182         axs[1].plot(list(range(len(self.valid_losses_list))), \
183                     self.valid_losses_list, "r--", \
184                     label = "Valid loss")
```

```
185         axs[1].set_xlabel("Iteration number")
186         axs[1].set_ylabel("Losses (in logarithmic scale)")
187         axs[1].set_yscale("log")
188         axs[2].plot(list(range(len(self.train_accuracies_list))), \
189                     self.train_accuracies_list, "b--", \
190                     label = "Train Acc")
191         axs[2].plot(list(range(len(self.valid_accuracies_list))), \
192                     self.valid_accuracies_list, "r--", \
193                     label = "Valid Acc")
194         axs[2].set_xlabel("Iteration number")
195         axs[2].set_ylabel("Accuracies")
196         # Display
197         axs[0].legend(loc = "best")
198         axs[1].legend(loc = "best")
199         axs[2].legend(loc = "best")
200         plt.show()
```

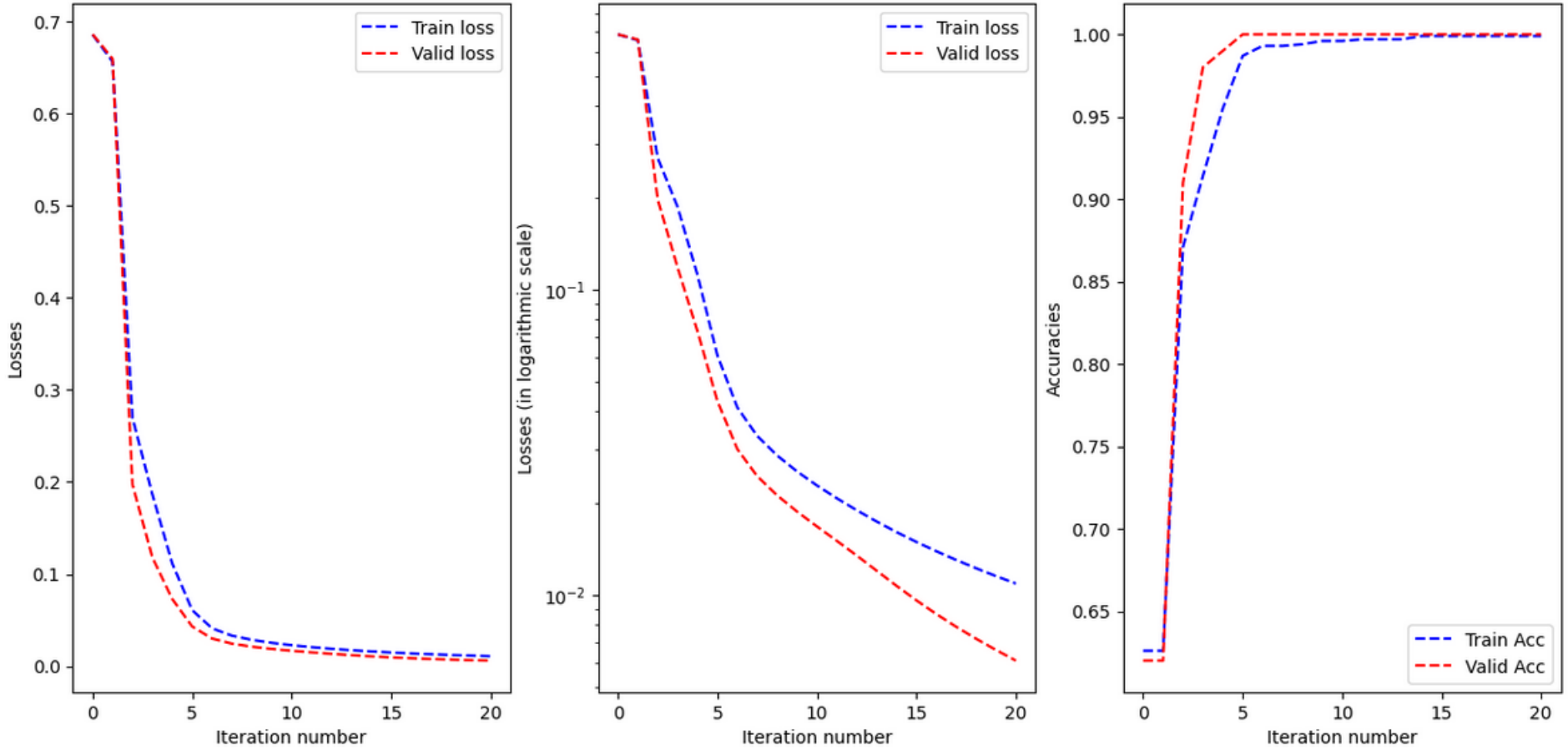
Good Practice #2: Using a train-test-valid split

Training our model gives following training curves (Good training).



Good Practice #2: Using a train-test-valid split

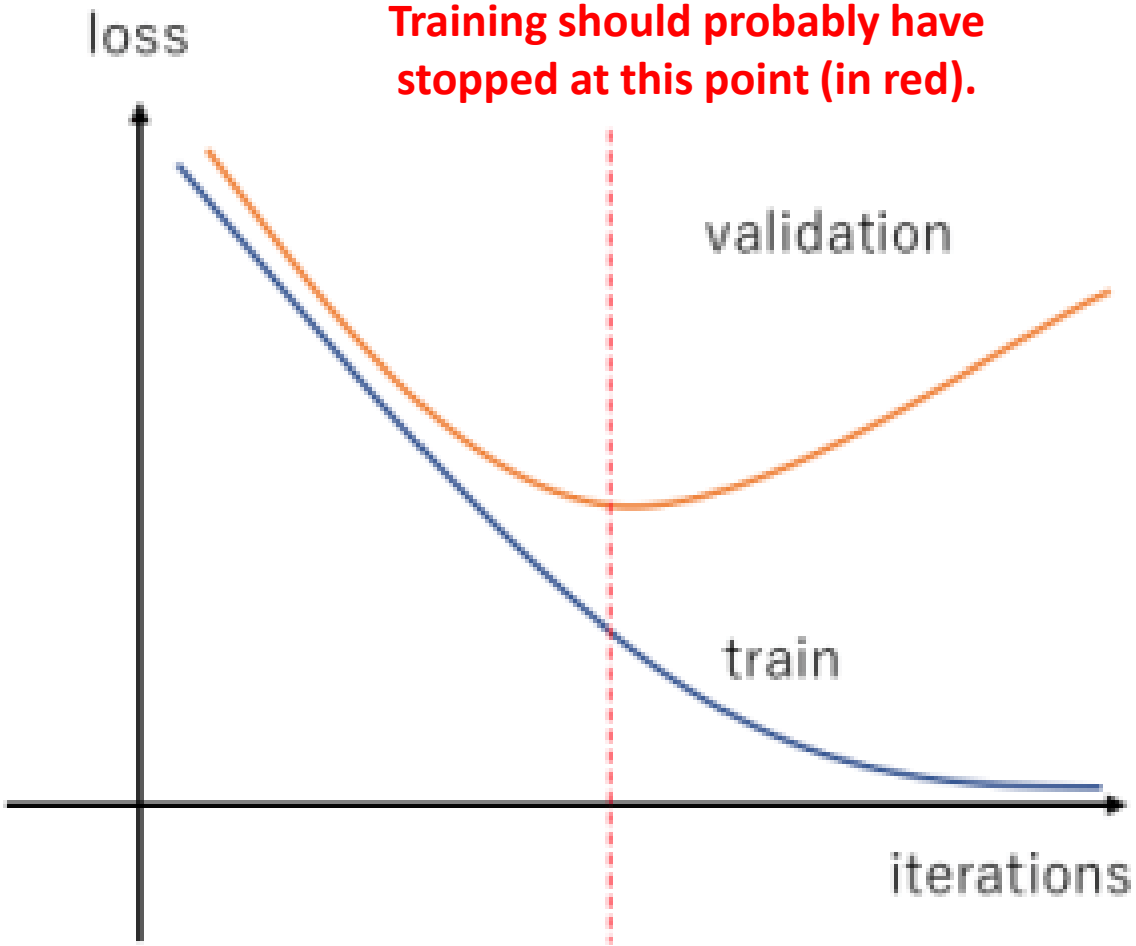
Note: A model that can generalize well will often have close curves on both training and validation sets. And a good final test accuracy value.



Good Practice #2: Using a train-test-valid split

In general,

- Model will start by **underfitting** the data, which is normal.
- After a few rounds of training models will train and (hopefully) achieve **good generalization**.
- Then, if training is pursued, the model will often attempt to **minimize loss at all costs**, **often sacrificing generalization (!)** in the process, and **overfitting.**



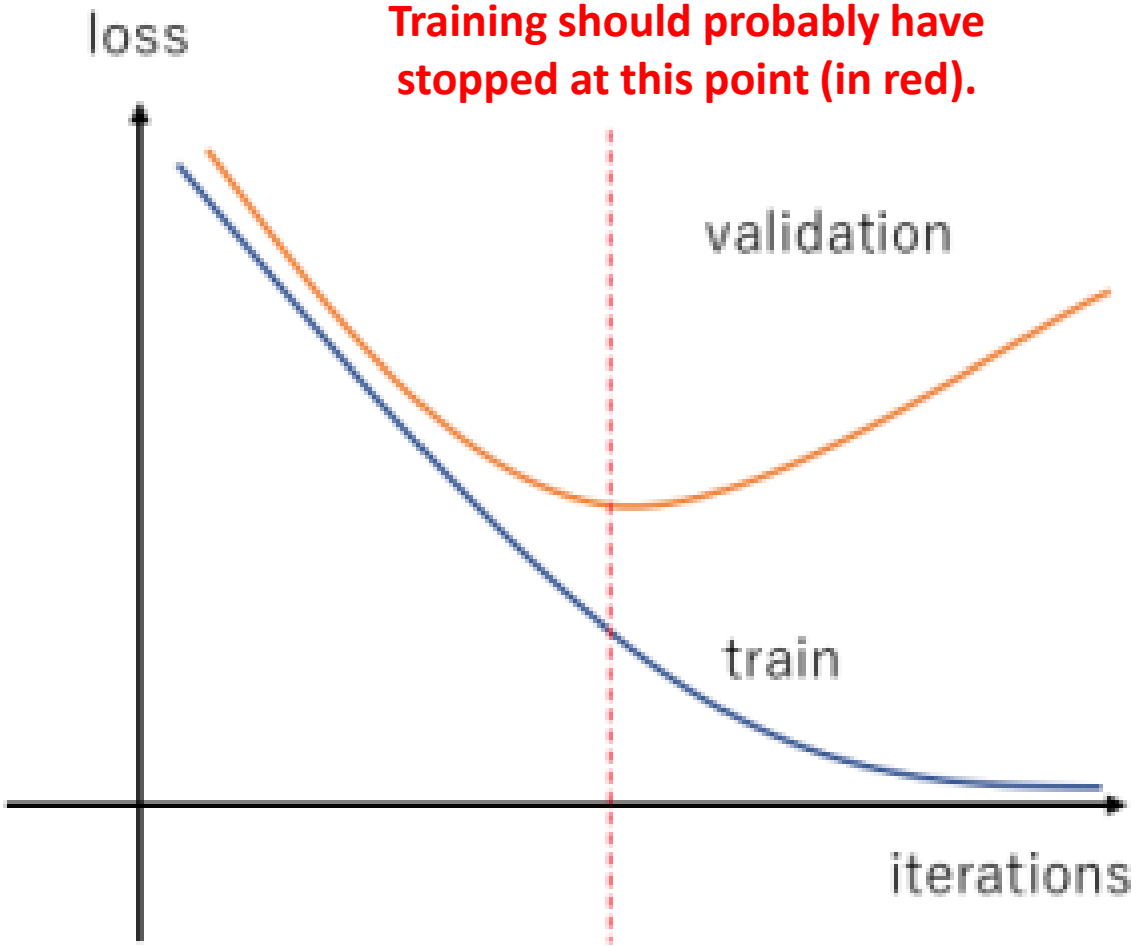
Good Practice #3: Implementing an early stop

Good Practice #3: Implementing an early stop.

It is often a good idea to try and **stop the training when the model starts losing its generalization capabilities and starts overfitting**.

We often want to stop when:

- Validation metrics achieve the best results (lowest loss or highest accuracy).
- Validation and training curves start going in opposite ways.



Good Practice #3: Implementing an early stop

Some approaches suggest to simply track a validation metric (e.g. accuracy) and define an **early stopping**, which interrupts training when:

- A high level of accuracy has been obtained (say, at least 98%),
- And the accuracy is no longer increasing (i.e. the change between two consecutive values of the accuracy is falling below a threshold).

```
# Check for delta value and early stop criterion
difference = abs(self.valid_accuracies_list[-1] - self.valid_accuracies_list[-2])
if(difference < delta and self.valid_accuracies_list[-1] > 0.98):
    if(display):
        message = "Stopping early - obtained good daccuracy (at least 98%)"
        message += " and accuracy evolution was less than delta on"
        message += "iteration {}.".format(iteration_number)
        print(message)
        break
```

Good Practice #3: Implementing an early stop

Some approaches suggest to simply track a validation metric (e.g. accuracy) and define an **early stopping**, which interrupts training when:

- A high level of accuracy has been obtained (say, at least 98%),
- And the accuracy is no longer increasing (i.e. the change between two consecutive values of the accuracy is falling below a threshold).

Counter-argument/Problem: Even with **early stopping**, we will often stop AFTER it is too late, as we need to see that the model starts overfitting and losing generalization capabilities to recognize that we should have stopped earlier!

Good Practice #4: Saver and loader functions

Counter-argument/Problem: Even with **early stopping**, we will often stop AFTER it is too late, as we need to see that the model starts overfitting and losing generalization capabilities to recognize that we should have stopped earlier!

Idea: How about we **save** the model trainable parameters (W_1, W_2, b_1, b_2) to a file (along with the iteration number) every K iterations?

→ **Good Practice #4: Use saver and loader functions.**

Good Practice #4: Saver and loader functions

Idea: How about we **save** the model trainable parameters (W_1, W_2, b_1, b_2) to a file (along with the iteration number) every K iterations?

- For simplicity, can use pickle.
- And dump our trainable parameters into some files.

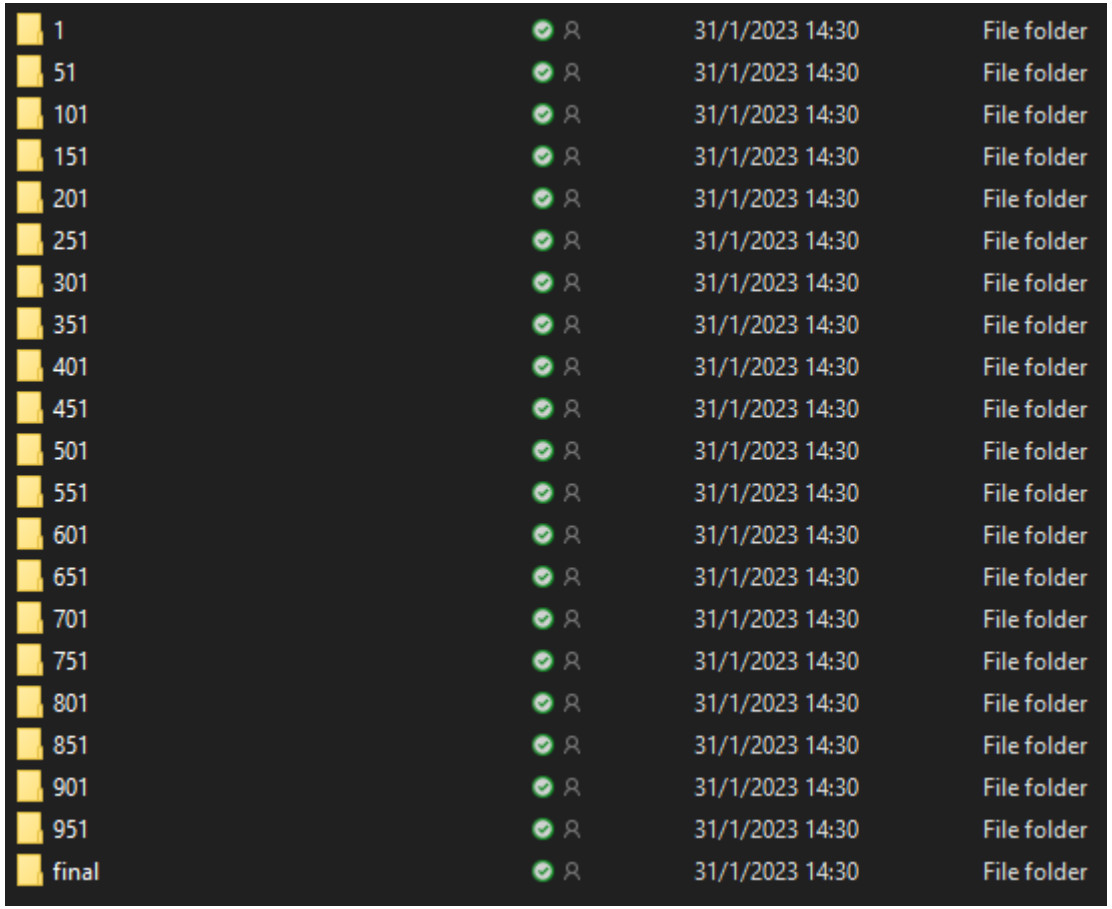
```python
def save(self, path_to_file, iter_num = "final"):
    # Display
    folder = path_to_file + "/" + iter_num + "/"
    print("Saving model to", folder)

    # Check if directory exists
    if(not os.path.exists(folder)):
        os.mkdir(folder)

    # Dump
    with open(folder + "W1.pkl", 'wb') as f:
        pickle.dump(self.W1, f)
    f.close()
    with open(folder + "W2.pkl", 'wb') as f:
        pickle.dump(self.W2, f)
    f.close()
    with open(folder + "b1.pkl", 'wb') as f:
        pickle.dump(self.b1, f)
    f.close()
    with open(folder + "b2.pkl", 'wb') as f:
        pickle.dump(self.b2, f)
    f.close()
```

```python
# Save model
self.save("./save", iter_num = str(iteration_number))
```

Good Practice #4: Saver and loader functions

Idea: How about we **save** the model trainable parameters (W_1, W_2, b_1, b_2) to a file (along with the iteration number) every K iterations?

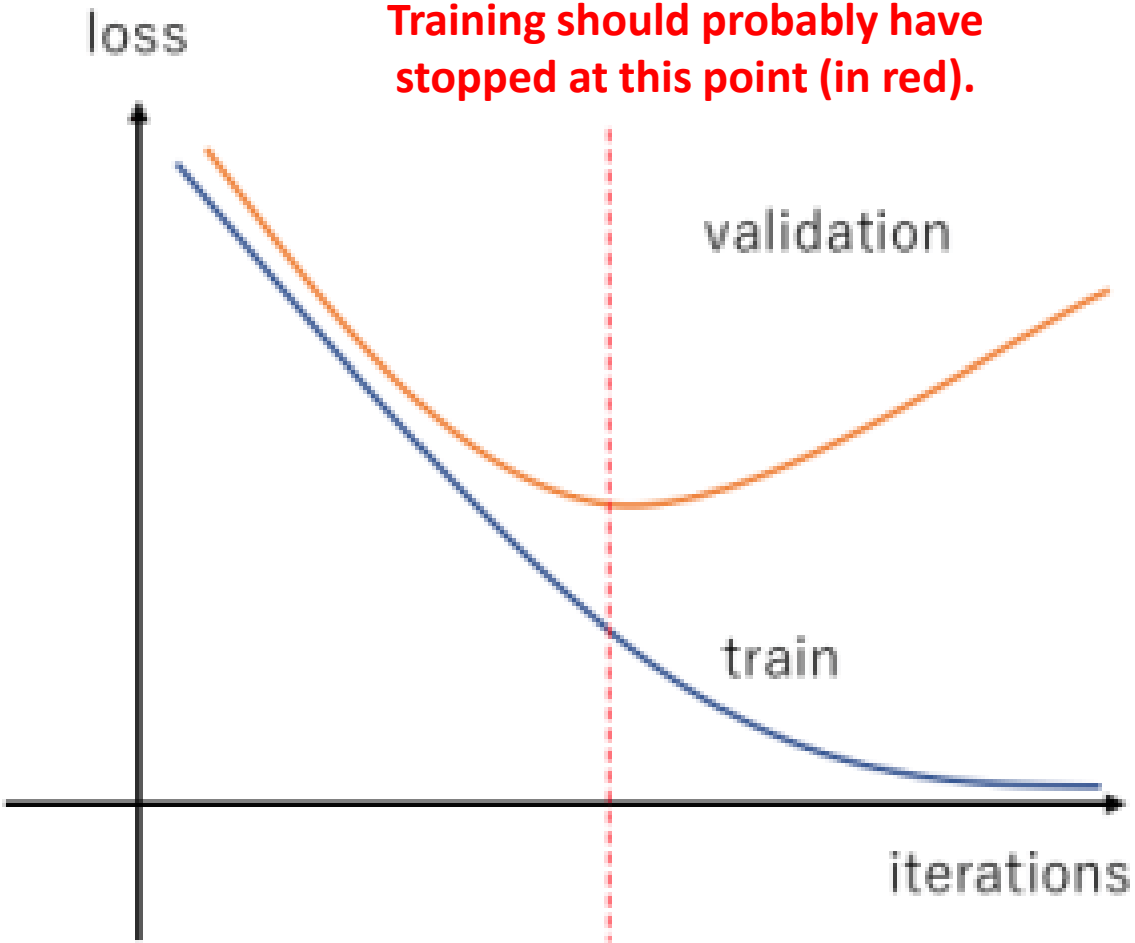
- For simplicity, can use pickle.
- And dump our trainable parameters into some files.



Good Practice #4: Saver and loader functions

Idea: How about we **save** the model trainable parameters (W_1, W_2, b_1, b_2) to a file (along with the iteration number) every K iterations?

- For simplicity, can use pickle.
- And dump our trainable parameters into some files.
- Can then load the best parameters after training!



Training should probably have stopped at this point (in red).

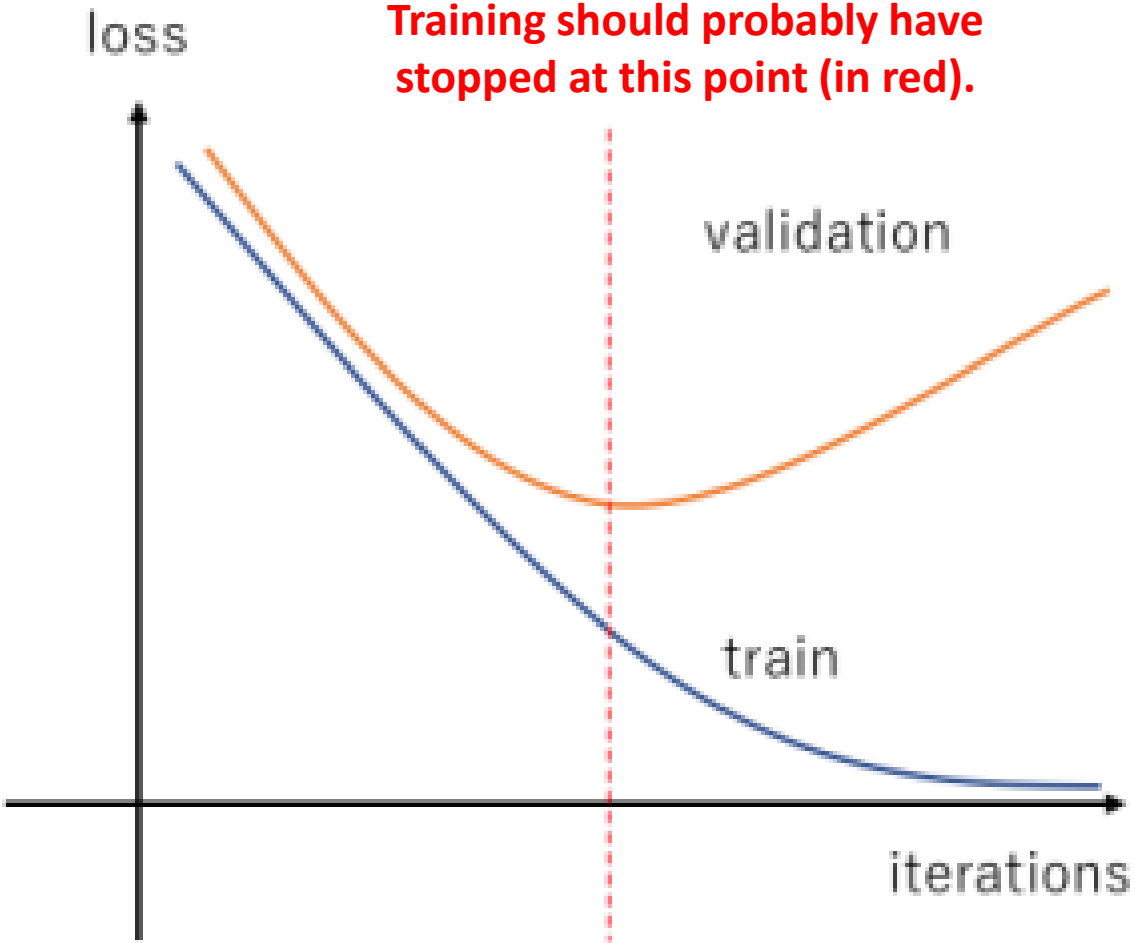
Trained for too long? No big deal, backtrack and load the weights corresponding to the best iteration!

Good Practice #4: Saver and loader functions

Good Practice #4: use saver and loader functions.

After training, look at your training curves and **load** the model parameters (W_1, W_2, b_1, b_2) using the files from the iteration corresponding to the best one for the model in terms of **generalization**!

(Iteration number to be decided manually.)



Training should probably have stopped at this point (in red).

Trained for too long? No big deal, backtrack and load the weights corresponding to the best iteration!

Good Practice #4: saver and loader functions

Good Practice #4: use saver and loader functions.

After training, look at your training curves and **load** the model parameters (W_1, W_2, b_1, b_2) using the files from the iteration corresponding to the best one for the model in terms of **generalization**!

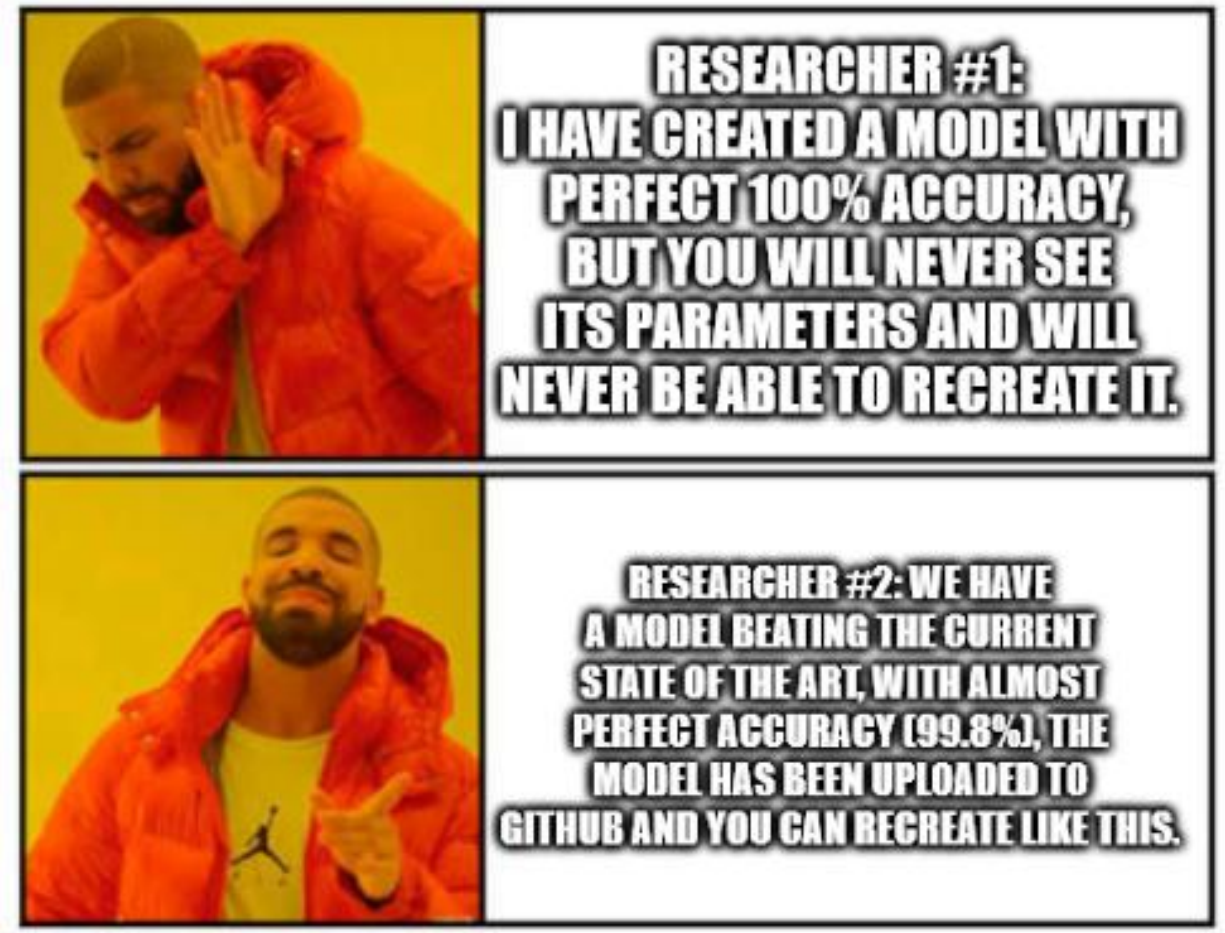
(Iteration number to be decided manually.)

```
def load(self, path_to_file, iter_num = "final"):
    folder = path_to_file + "/" + iter_num + "/"
    print("Loading model from", folder)

    # Load
    with open(folder + "W1.pkl", 'rb') as f:
        self.W1 = pickle.load(f)
    f.close()
    with open(folder + "W2.pkl", 'rb') as f:
        self.W2 = pickle.load(f)
    f.close()
    with open(folder + "b1.pkl", 'rb') as f:
        self.b1 = pickle.load(f)
    f.close()
    with open(folder + "b2.pkl", 'rb') as f:
        self.b2 = pickle.load(f)
    f.close()
```

Saver/loader functions are critical in AI

Reproducibility: After training, it is important that you give your client a simple way to load your trained model. You would not want the client to re-run the training (especially if it takes a VERY long time to train!)

It is also important if you do AI research, as your research will have little credibility if you are not able to let other people see the model at work.

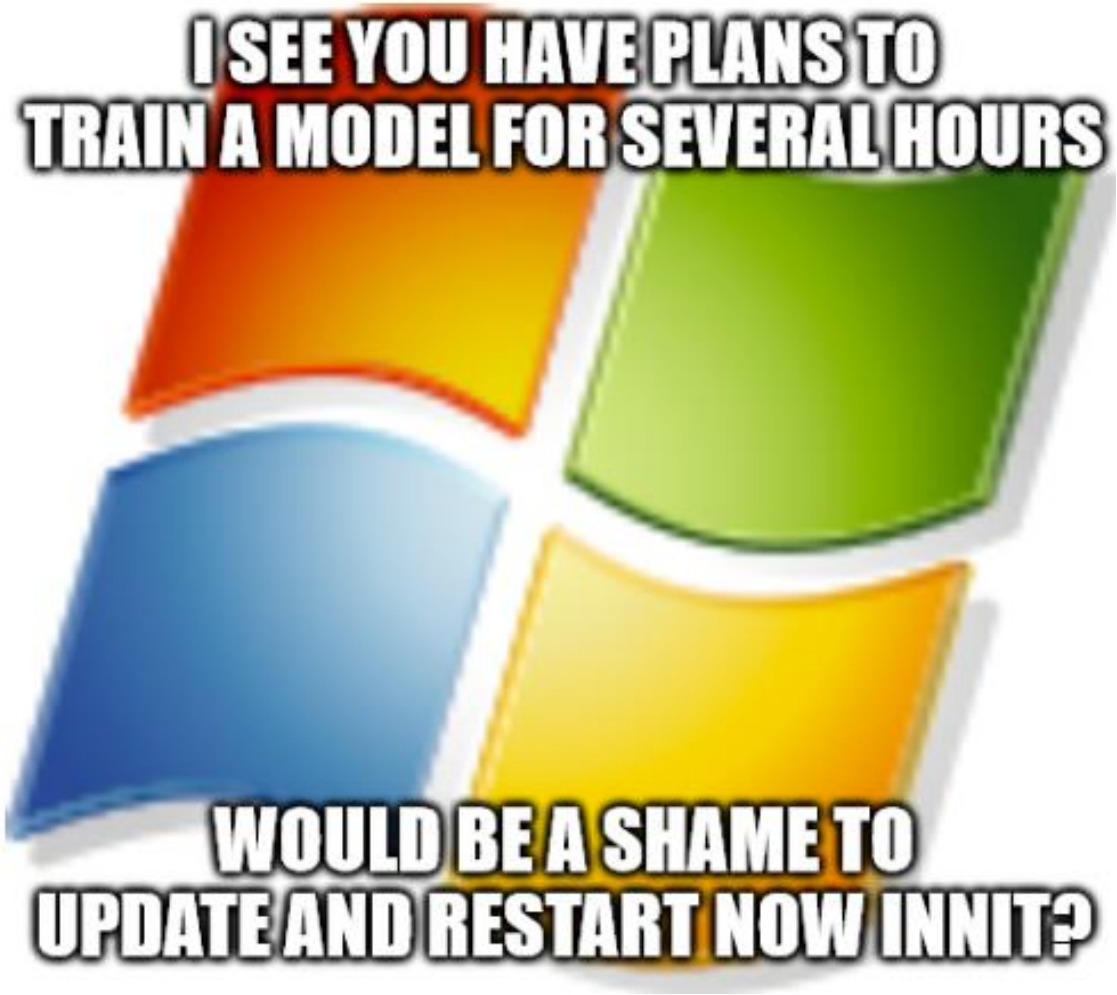


Saver/loader functions are critical

Preventing data loss: Training a heavy model can take a long time, sometimes even days or weeks, if the model architecture is massive.

If the training process is interrupted, all the progress made so far will be lost, unless the model has been saved.

In case of an interruption, we can simply load the latest save of the model, and resume the training, instead of restarting from scratch.



Typical questions about model architecture

Typical questions asked by students regarding the size of a neural network model.

- **How many layers does my model need?**
- **How many neurons should I put in each layer?**
- **Is it better to have more layers or more neurons per layer?**

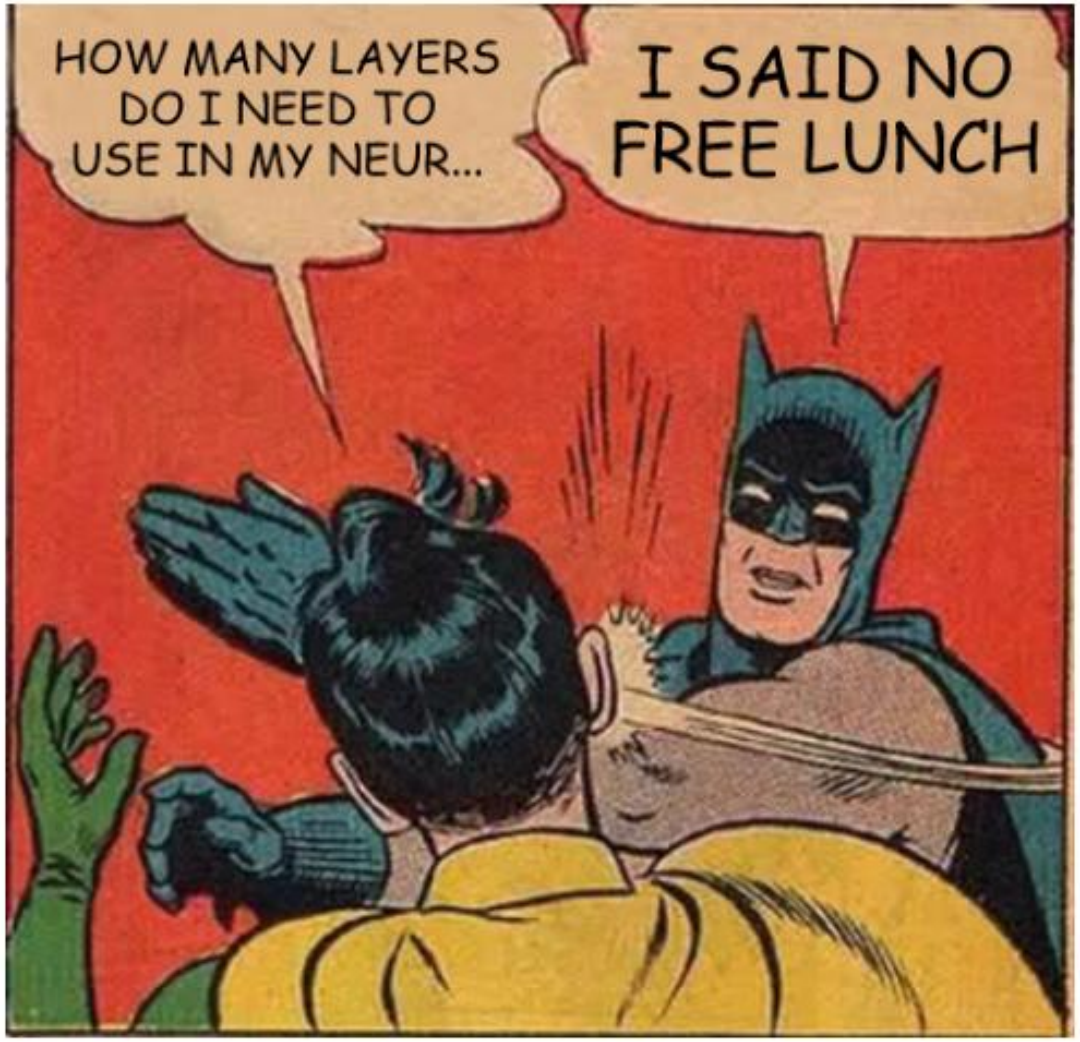
The answer to these questions will surprise you.

Typical questions about model architecture

Typical questions asked by students regarding the size of a neural network model.

- **How many layers does my model need?**
- **How many neurons should I put in each layer?**
- **Is it better to have more layers or more neurons per layer?**

The answer to these questions will surprise you.

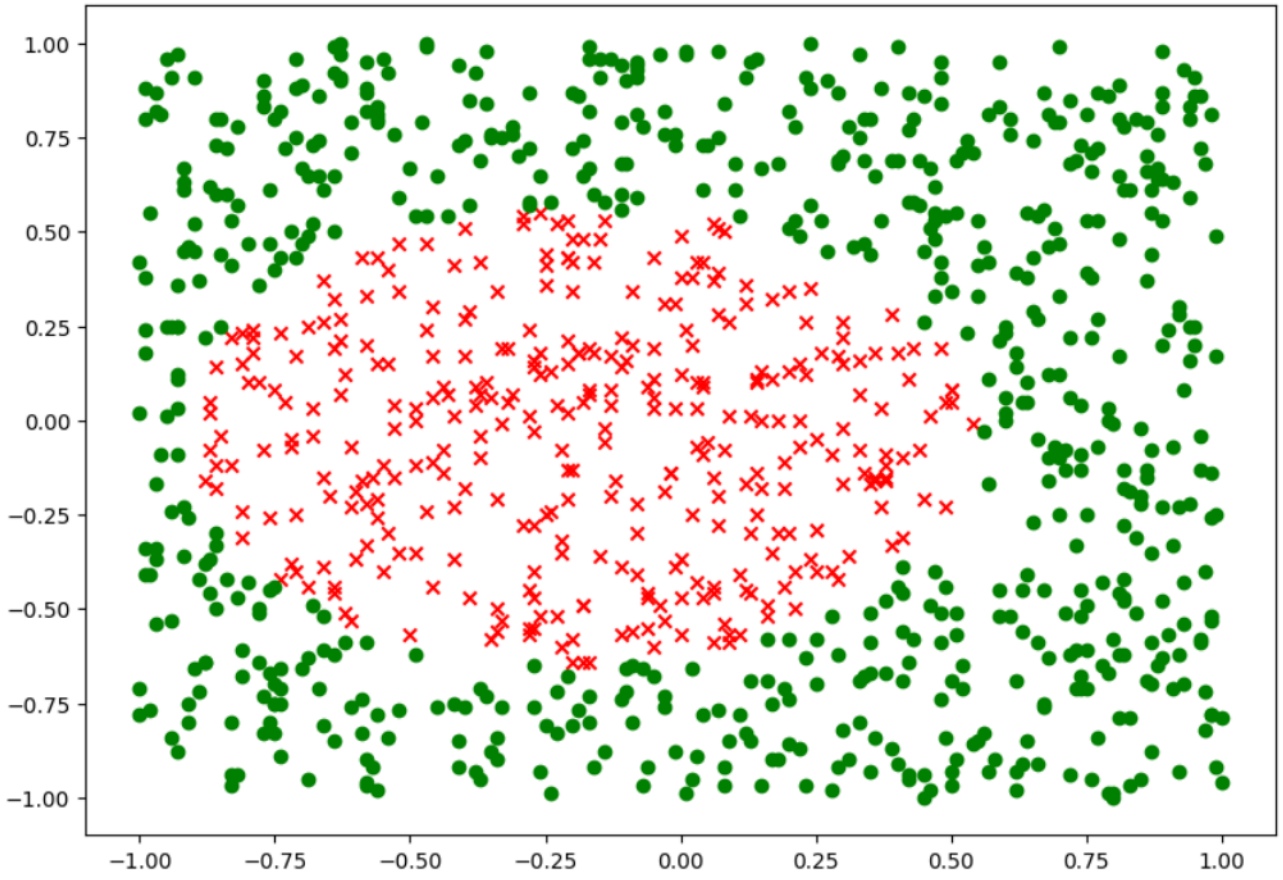


Problem statement

Consider the following binary classification dataset.

It follows the same logic as our previous dataset, but the boundary seems to have a different shape, and its equation is not given this time.

It looks nicely separable, so we should be able to train a classifier model with near-perfect accuracy.



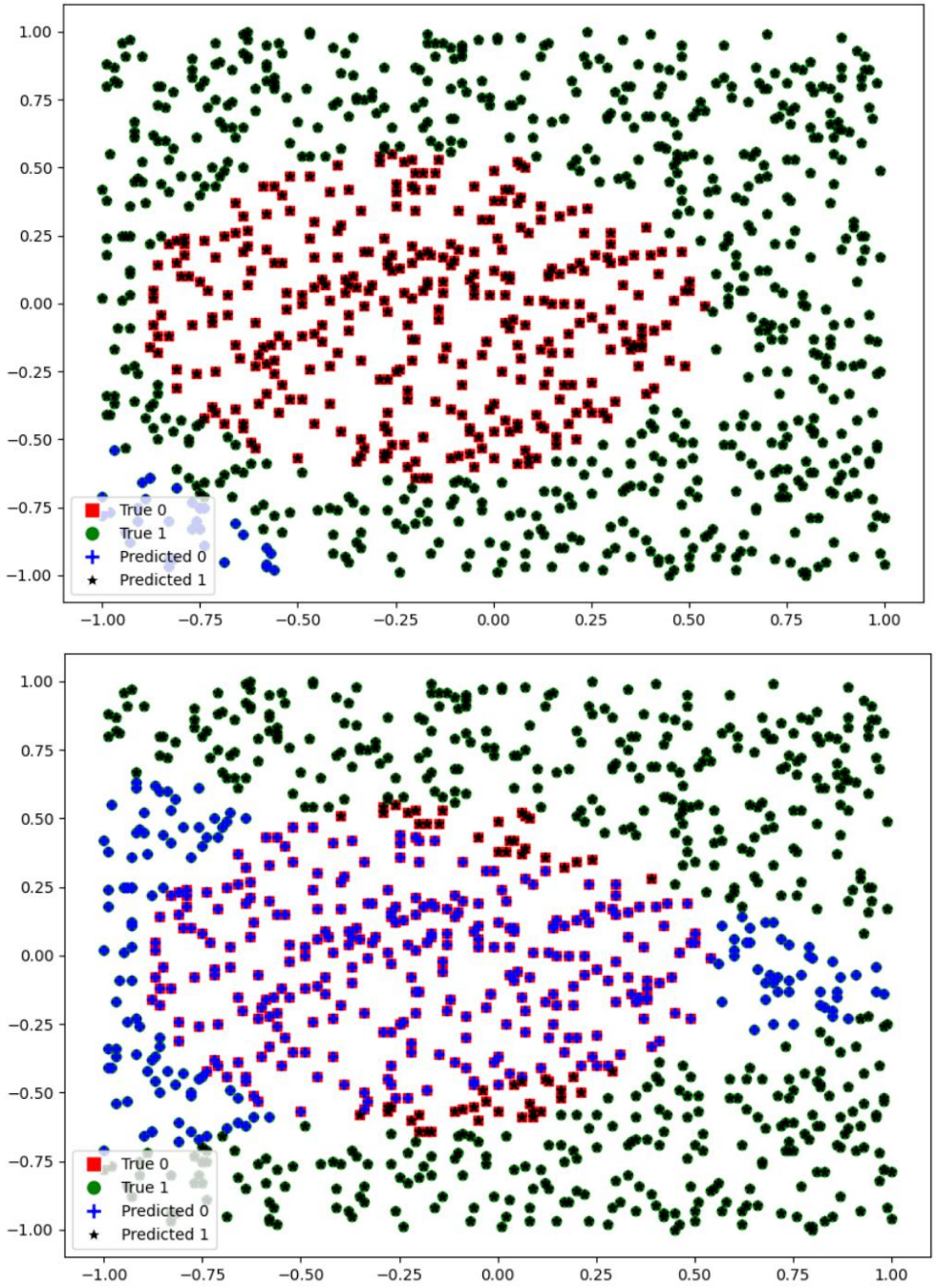
Problem statement

In Notebook 8 we will first try train two models:

- **Model 1:** 1-layer neural network, 62% accuracy (on top)
- **Model 2:** 2-layers neural network, 81% accuracy (bottom).

Unfortunately, it seems both models are struggling to learn…

Question: What could we do then?



Problem statement

Three suggested approaches/tasks for you to try in this Notebook 8:

1. Adding a second layer helped, let us try **adding a third layer**?
   (We will need to rework most methods in our Neural Network class, as it only includes two layers at the moment).
2. How about we stick to two layers only and **use more neurons in the first layer** of the neural network?
3. How about a bit of **feature engineering**?
   (That is reworking our dataset a bit to produce more useful features that could help our neural networks)

Guided Practice, solutions are provided (but no cheating!)

Well, let us experiment then?

Have a look at Notebook 8, it comes with three practice tasks.
(Solutions are also available in the solutions folder).

No homework again this week, so playing with this notebook is your homework!

Lesson #1: Features engineering is king!

In general**, feature engineering** is the direction which offers the largest improvement potential for our models.

If you have a hunch about which features could help the network classify better, you should attempt some **feature engineering**.

Problem: it requires a strong mathematical intuition/domain expertise about the dataset!

In Notebook 8, what helped was recognizing that the squared values of the inputs could help. This followed from seeing that the boundary looked like a **circle** and that **circles equations** are usually relying on **squared values of the inputs**.

$$x_1^2 + x_2^2 = K$$

Our boundary equation for the dataset was in fact an **ellipse** equation:

$$2x_1^2 + 3x_2^2 + 0.75x_1 + 0.25x_2 = 0$$

Lesson #2: Larger layers vs. More layers

The number of layers to use in a neural network can vary depending on the complexity of the task and the size of the data. It is a hyperparameter that can be tuned through experimentation.

- **In general, deeper networks (more layers) can learn more complex representations, but they are also more computationally expensive and prone to overfitting.**
- **On the other hand, shallower networks are less expensive and less prone to overfitting, but they may not have enough capacity to learn the task.**

Lesson #2: Larger layers vs. More layers

As far as I know, there is no rule of thumb formula for determining the number of layers to use based on the shape of a dataset.

The number of layers in a neural network model is determined by a variety of factors, including the complexity of the problem, the size of the input data, and the available computational resources.

1. If the data is linearly separable then you do not need any hidden layers or activation functions, just use a linear regression!
2. If data is low-complexity and has few dimensions/features, then 2 layers would often work, or are a good starting point.
3. If data is high-complexity or has many dimensions/features, then using more hidden layers is required (ideal number to be tested).

Lesson #2: Larger layers vs. More layers

However, “adding an extra layer to a model that is failing and already has many layers in the hopes that it will suddenly fix it” is often foolish and will have limited results.

You need enough layers to match the complexity of the task, anything beyond that is most likely overkill and will not help.

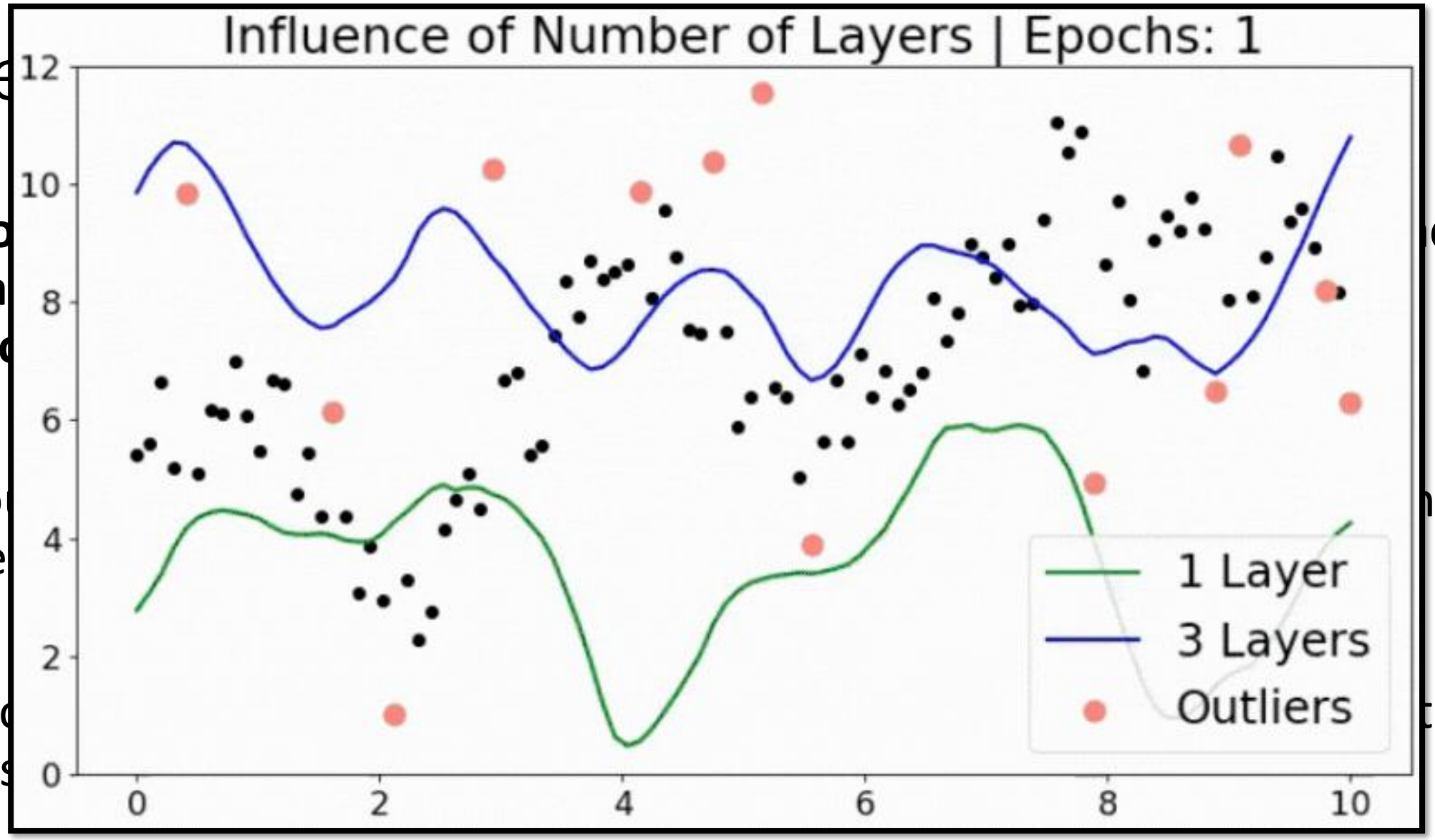
More often than not, you are NOT using the right **TYPE** of layers that fits the **TYPE** of data you are playing with (more on this later).

Le

Ho **dy**
ha
foc

Yo ng
be

Mo t
fits



In fact, adding more layers will likely lead to more overfitting!

Lesson #2: Larger layers vs. More layers

In fact, it is often **more beneficial to add more neurons to a layer** (as you might have seen in task #3). As far as I know, there is no "ideal" mathematical formula for the number of neurons to use in a layer.

But some people ([StackNN]) have been experimenting.

A good rule of thumb could be to have a number of neurons in the first layer, N_n, such that:

$$N_n = \frac{N_s}{(N_i + N_o)\gamma}$$

With N_s the number of samples in dataset, N_i the number of inputs (or inputs dimensionality), N_o the outputs dimensionality and γ a value between 2 and 20.

Lesson #2: Larger layers vs. More layers

After the number of neurons for the first layer N_{h_1} has been decided, it is often commonly accepted to have the number of neurons:

- **Be a power of two (for binary reasons, except for inputs and outputs, where you should have what the dataset requires)**
- **And progressively decrease, for instance dividing by two (or more) the number of neurons after each layer.**

$$N_x = 4 \rightarrow N_{h_1} = 16 \rightarrow N_{h_2} = 8 \rightarrow N_{h_3} = 4 \rightarrow N_y = 1$$

These are not absolute rules (ever heard of NFL?), however, so go ahead and experiment!

A friendly reminder, though, in case you still do not get it

Conclusion (Week 2)

- The no free lunch theorem, again
- Train-test-validation split
- Early stopping
- Saver and loader functions
- A few more commonly accepted good practices

Next week?

- Introduction to PyTorch framework and tensors
- Implementing shallow Neural Networks in PyTorch
- The power of AutoGrad and CUDA processing
- Optimizers, initializers in PyTorch
- Datasets and dataloaders
- From shallow to deep NNs

Hyperparameters tuning and searching

Definition (hyperparameters tuning and searching):

Earlier, we have identified that several parameters in our neural networks had to be manually decided by the user and called those **hyperparameters**.

Unfortunately, the **NFL theorem** tells us there is often **no closed-form formula** to tell you what is the **best value to use for those hyperparameters**.

Typical **hyperparameters** include:

- Number of layers and their sizes,
- Initializers to use for parameters,
- Activation functions to use,
- Learning rate, momentum and other parameters related to optimizers,
- Etc.

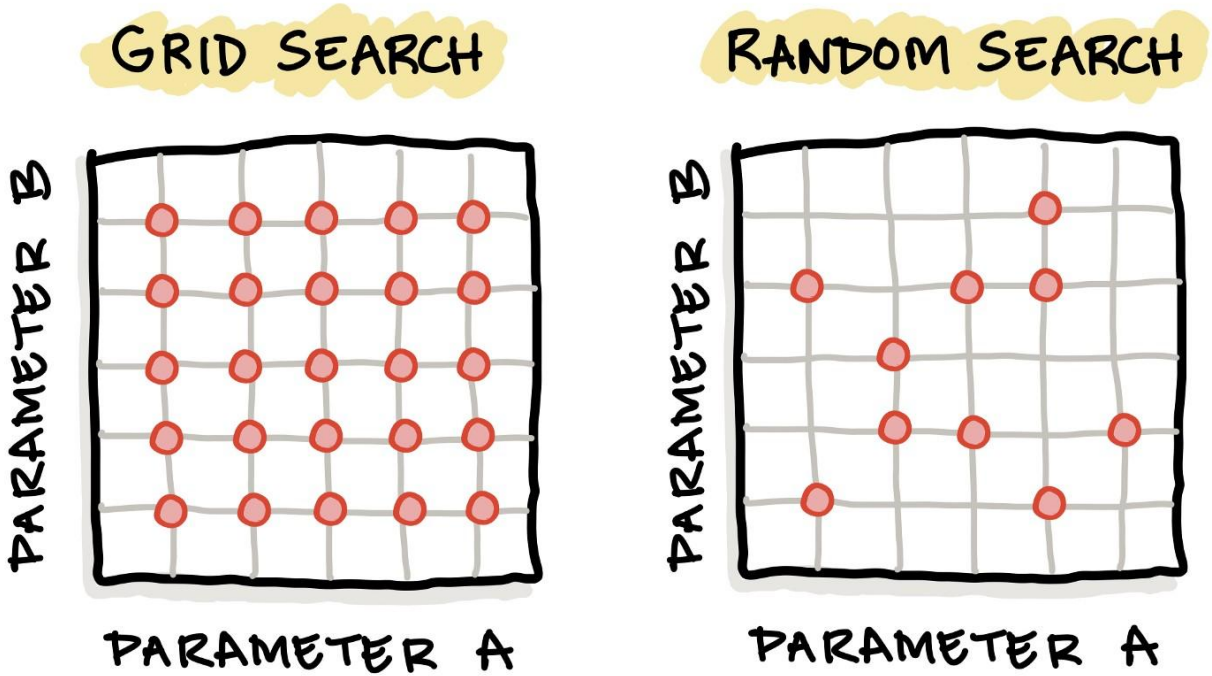
No choice: try different values and possibilities. **How to search?**

Hyperparameters tuning and searching

Definition (hyperparameter grid searching):

Grid search is **the simplest algorithm for hyperparameter tuning**.

Basically, we divide the domain of possible values for each of the hyperparameters into a discrete grid. Then, we will try every combination of values of this grid, calculating some performance metrics.

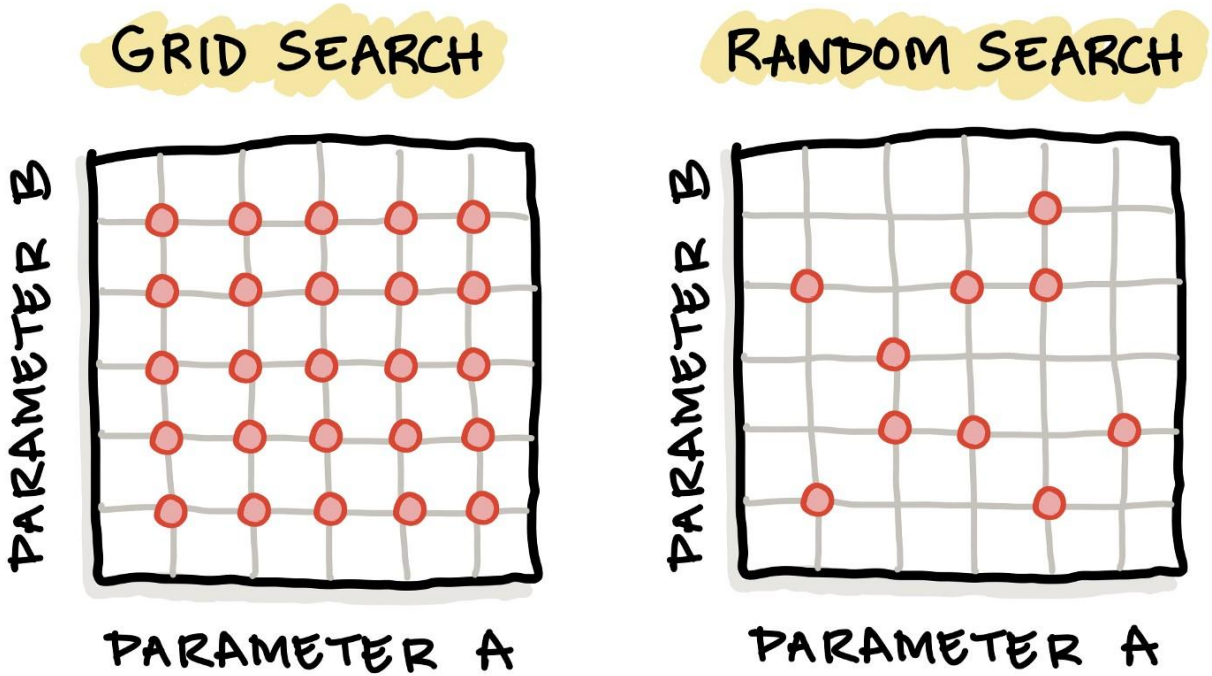


Hyperparameters tuning and searching

Definition (hyperparameter random searching):

Random search is **another algorithm for hyperparameter tuning**.

Random search defines a search space as a bounded domain of hyperparameter values and randomly sample points in that domain. After trying enough values, keep the best hyperparameters configuration.

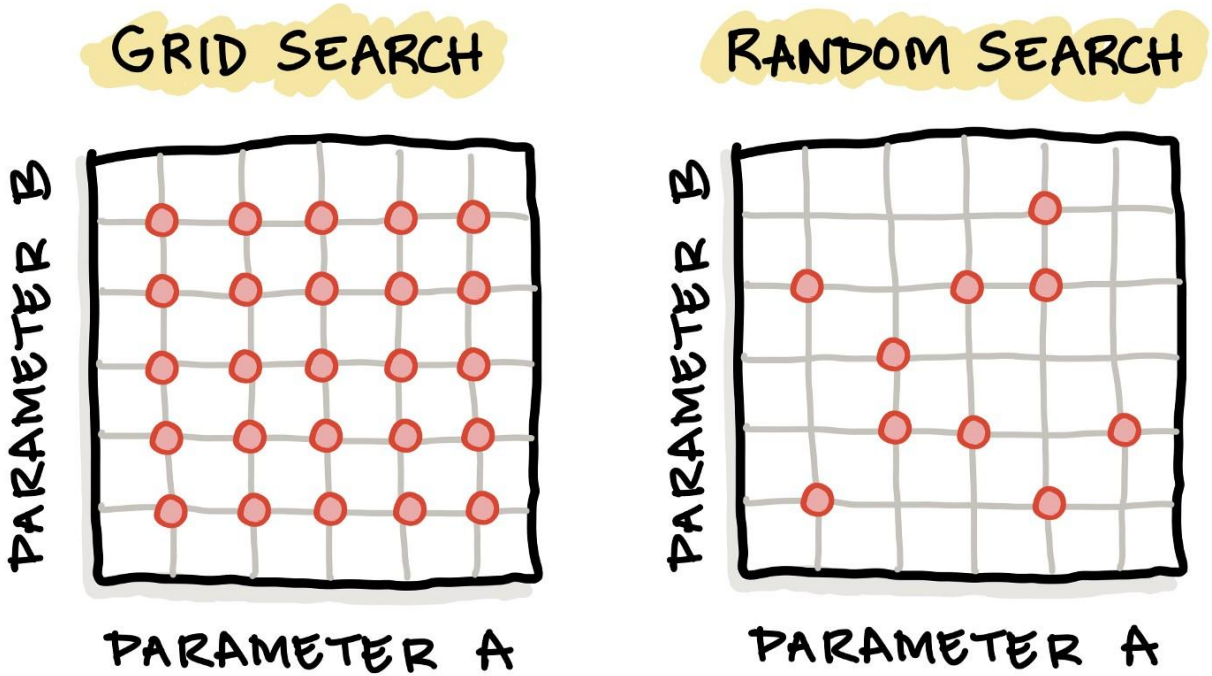


Hyperparameters tuning and searching

In general (see [HypSearch]),

- Grid search is often better on small spaces,
- Whereas random search is better on large domains but has no guarantee of returning a good configuration.

More advanced techniques beyond these two exist, such as Bayesian Optimization, Evolutionary Optimization, etc.



Hyperparameters tuning and searching

Your objective should be to find **the best values for all the hyperparameters** $\boldsymbol{P}^*$ to use in the model (learning rate, number of layers, number of neurons, regularization, etc.).

Your objective should then be to obtain the best possible value for the **validation** loss/accuracy value for the model, after you have loaded the best values for the trainable parameters (as in Good Practice #4).

The **test** loss/accuracy value is the one you can safely advertise to show **generalization**, as the validation metrics have technically been used to adjust hyperparameters (which is considered part of training).

→ Another optimization problem on our hands to experiment with!

Learn more about these topics

Some extra (easy) reading and videos for those of you who are curious.

- [HypSearch] How to run hyperparameters searching in Python, implementations using Scikit learn and more advanced methods.
  https://towardsdatascience.com/7-hyperparameter-optimization-techniques-every-data-scientist-should-know-12cdebe713da
  And
  https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/
- [StackNN] How to decide on a number of neurons and layers in Neural Networks?
  https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw