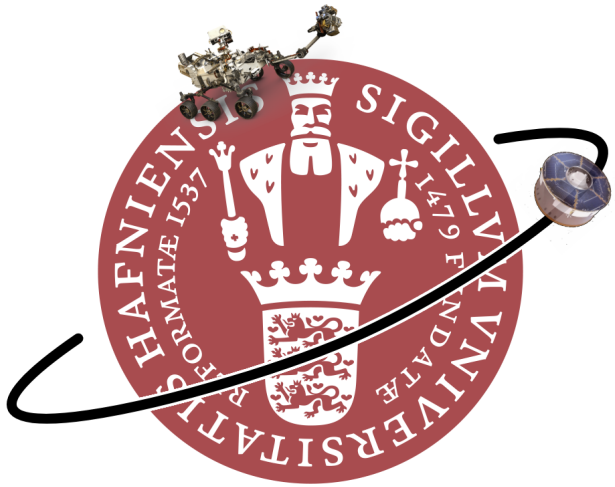
AML goes to Mars

Using supervised learning to calibrate LIBS data for the *ChemCam* emission spectrograph on the Mars Curiosity rover



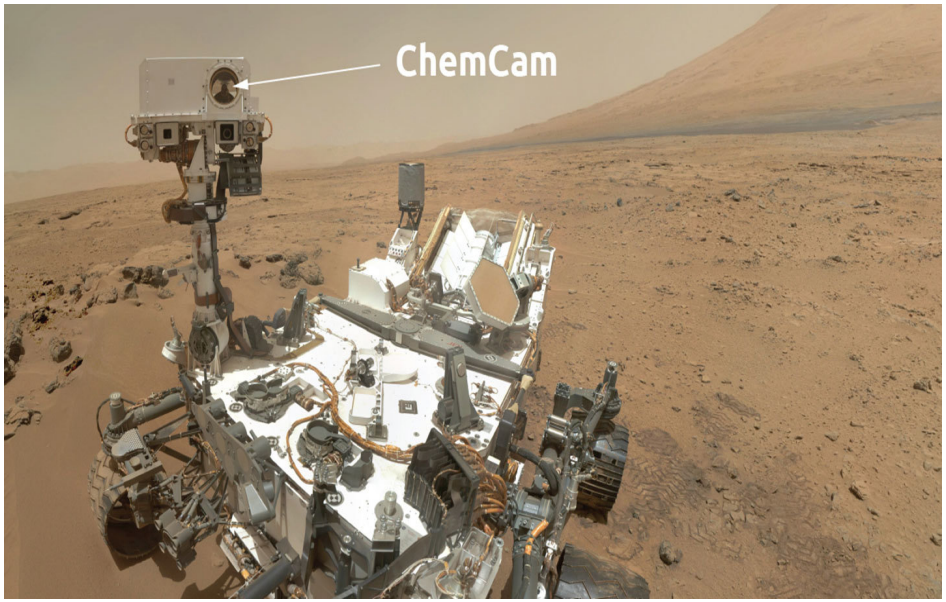
University of Copenhagen – UCPH
Niels Bohr Insititute – NBI

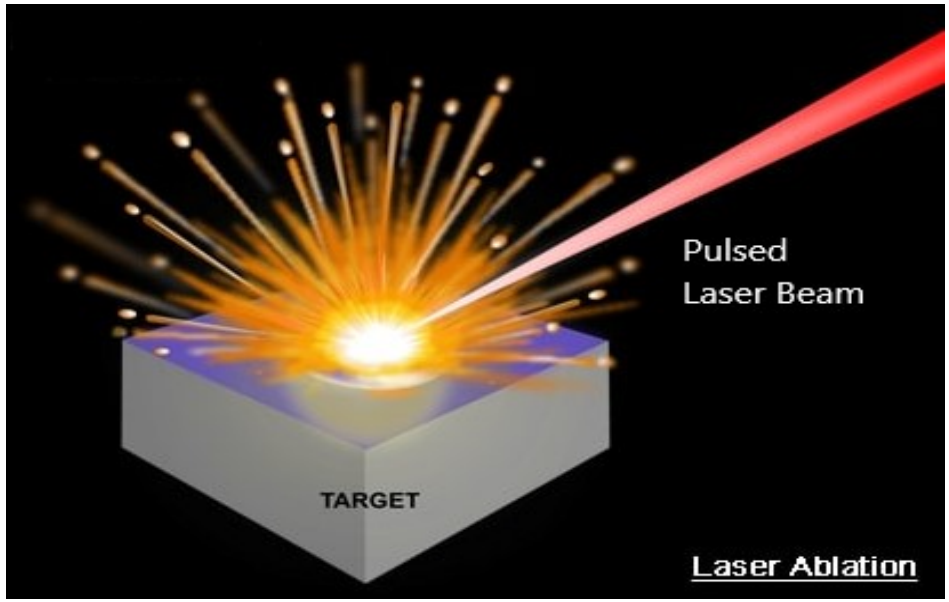
June 16, 2021

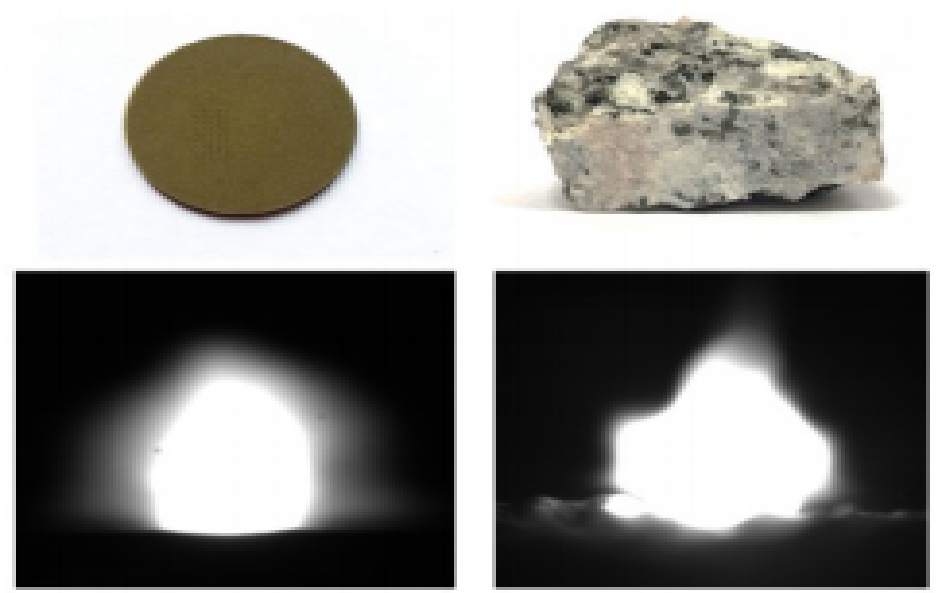
All members contributed equally in the making of this project
Morten, Kristian, Niall, Jonathan, Frederik & Leon

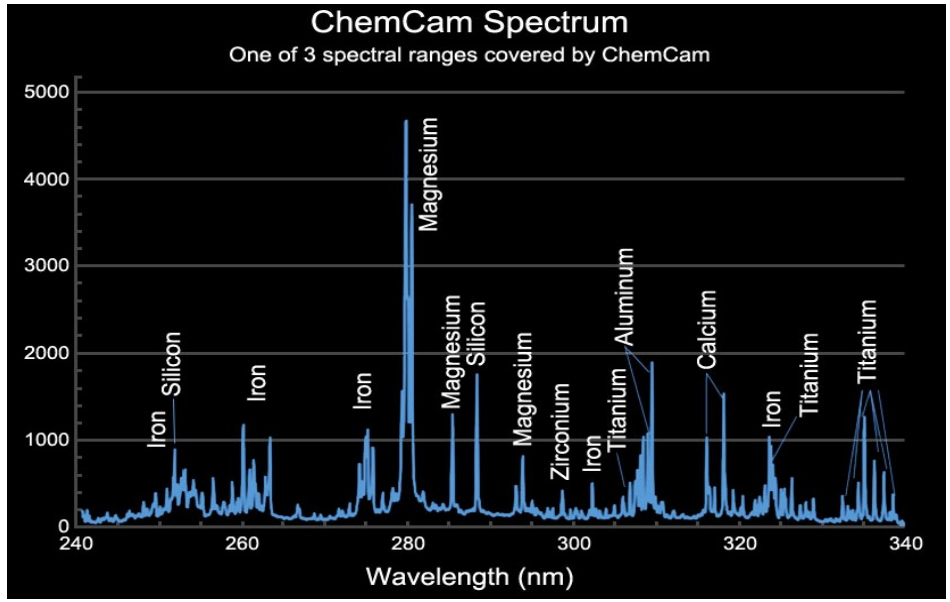
Contents

Intro: What is ChemCam





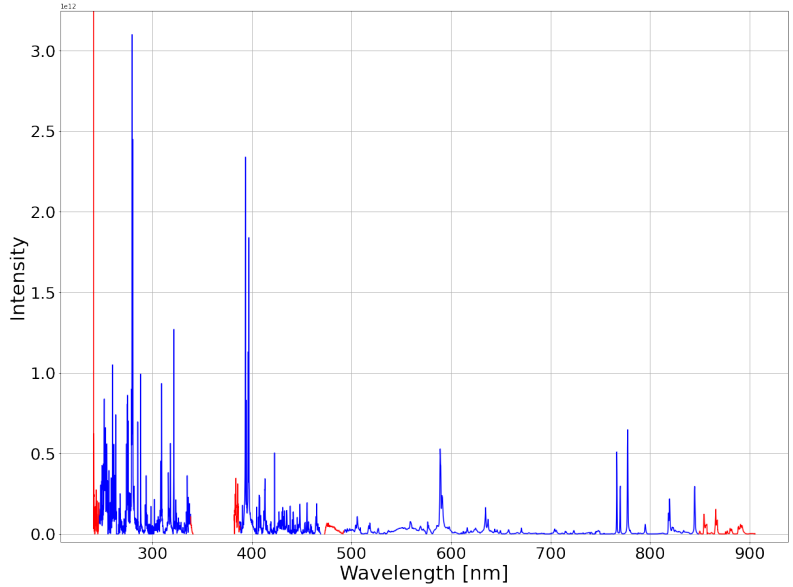




Credit: NASA/JPL-Caltech/LANL

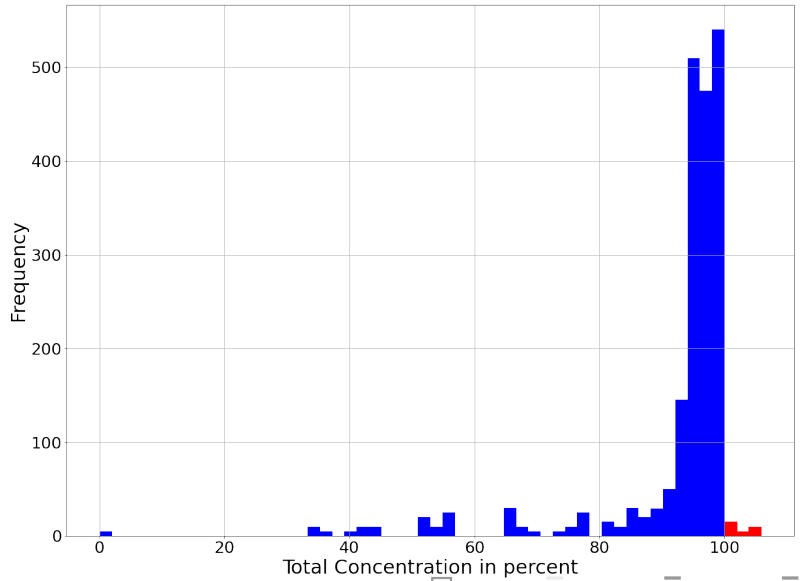
Data

- 408 samples with different chemical and mineral compositions
- 4-5 spectra per sample - 2039 in total
- Concentration of 9 different chemicals and minerals
  - SiO_2, TiO_2, Al_2O_3, $FeOT$, MnO, MgO, CaO, Na_2O and K_2O
- Intensity for 6144 wavelengths in the range 241-906 nm



Data Cleaning

- Outer wavelength layers
- Samples with different names and spectra, but identical compositions
- Total concentration above 100%
- Outliers - up to 18σ from mean intensity for wavelength



Final dataset:

- 1654 spectra with 5486 features

Normalisation and Scaling

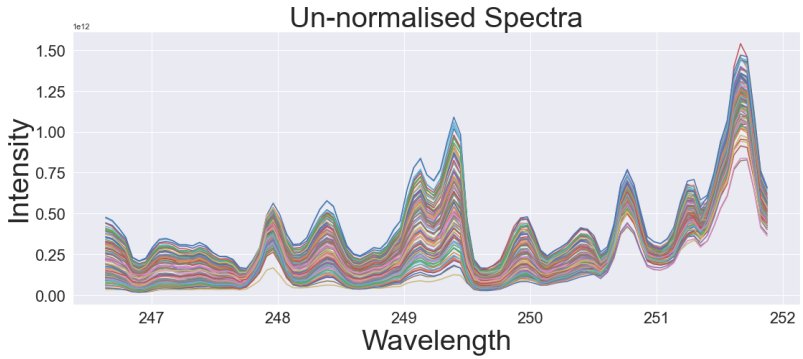
1. Normalisation: Spectra may be normalised individually (row-wise) - experimental fluctuations produce spectra of different scales due to e.g. plasma temperature
2. Standardisation: Wavelength channels may be standardised (column-wise) to give each channel zero mean and unit variance. Some ML algorithms expect Gaussian distribution of the features.
3. Quantitative scaling method test: Spectral data transformed with Scikit learn's 7 scaling methods, we find mean absolute error on percentage composition for DecisionTree and Keras multi layer perceptron.

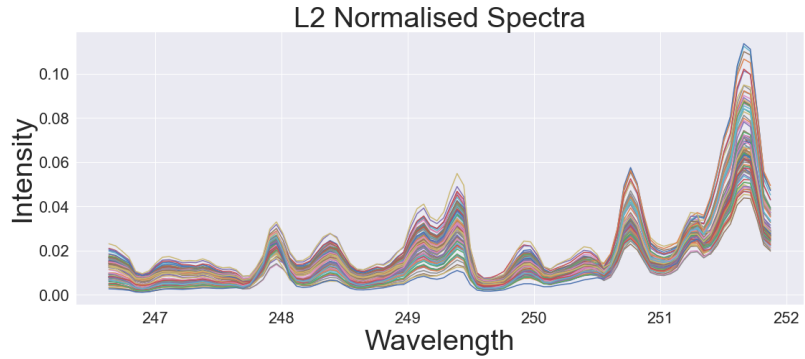
Scaler	DecisionTree MAE	MLP MAE
Unscaled (baseline)	0.957	744758.9
StandardScaler	0.946222	1.026088
RobustScaler	0.946883	3125.461
MinMaxScaler	0.987789	1.767521
MaxAbsScaler	0.958454	1.667965
Normalizer L2-norm	0.803676	1.870137
Normalizer L1-norm	0.820308	3.021888
Normalizer Max-norm	0.876773	1.831981
QuantileTransformer	0.971722	1.523372
PowerTransformer	0.93557	1.094539
Normalizer L2 and StandardScaler	0.803599	1.150329

Normalisation and Scaling II

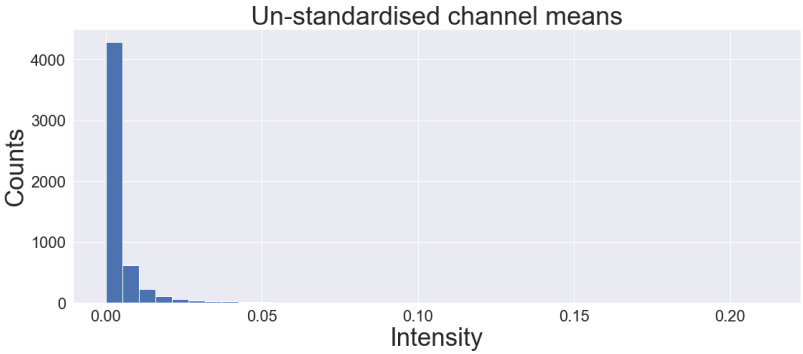
Best performer:

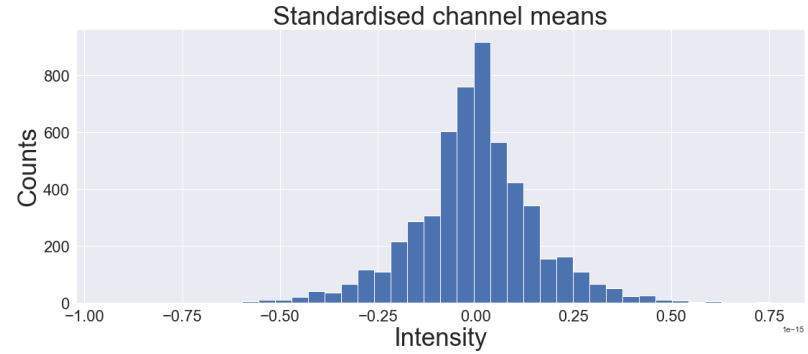
1. Normalisation: scikit-learn Normalizer - L2-norm weights: $w_i = \left(\sum_{j=1}^{n} x_{i,j}^2 \right)^{\frac{1}{2}}$





2. Standardisation: Standard normal variate (SNV) transformation with scikit-learn StandardScaler



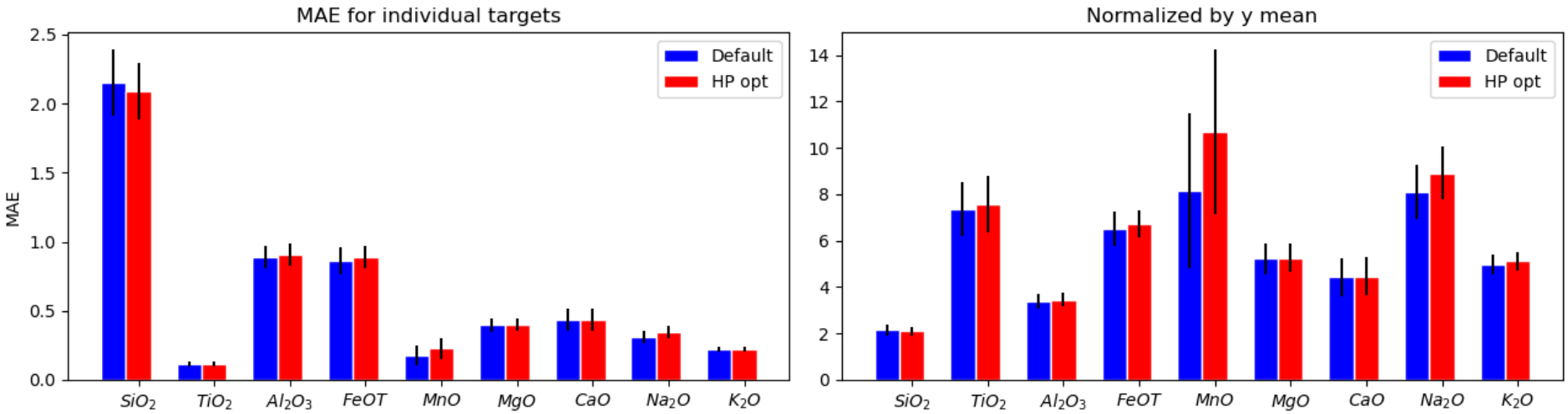


3. Final models: L2 Normalising improves LightGBM, VAE and CNN model performance, but standardisation appears to have negative impact. We proceed with only spectral normalisation.

Tree based models

Tree-based model can serve as an easy baseline.
Here we have chosen to go with LightGBM.

	Default settings	HP opt w. RandomSearch
MAE over all targets	0.62(0.08)	0.63(0.07)



Variational Autoencoder (VAE) model

$$\mathcal{L}_{total} = \mathcal{L}_{recon} + \mathcal{L}_{KL} + \mathcal{L}_{reg}$$

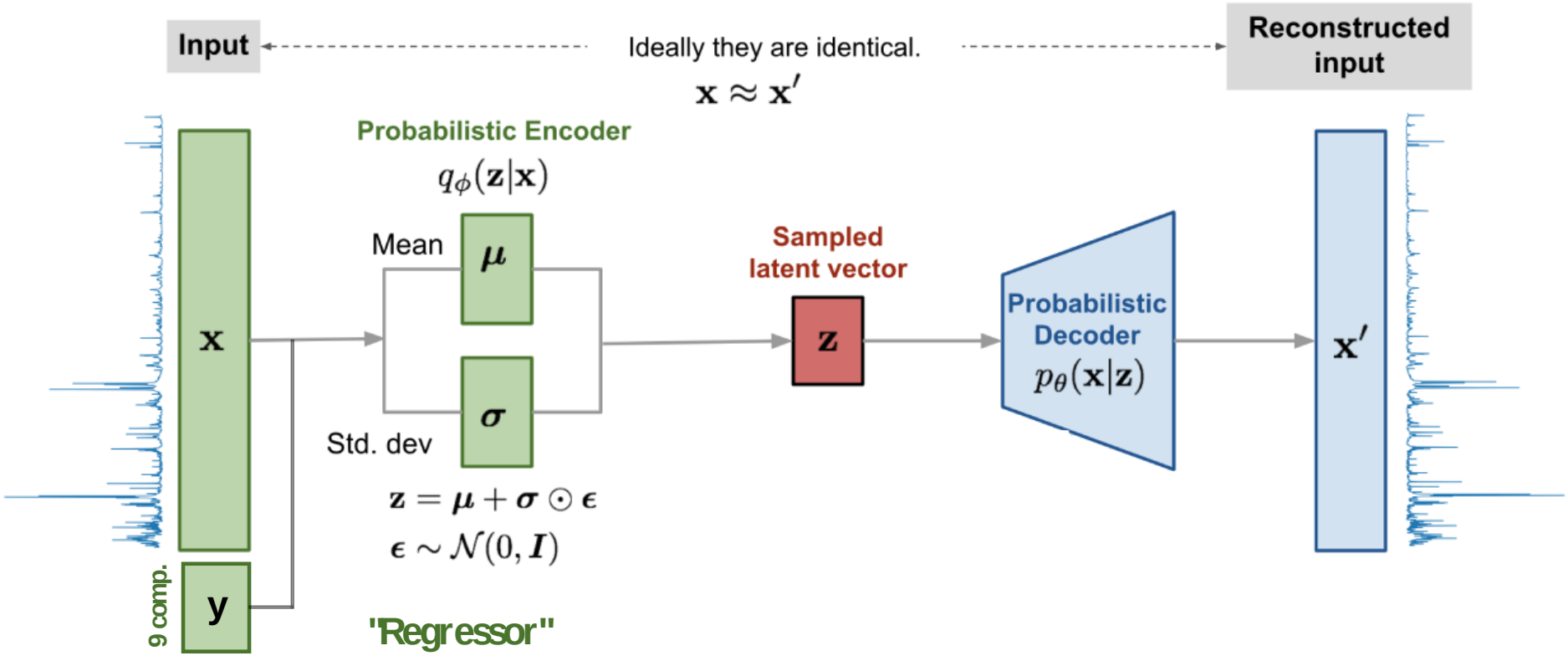
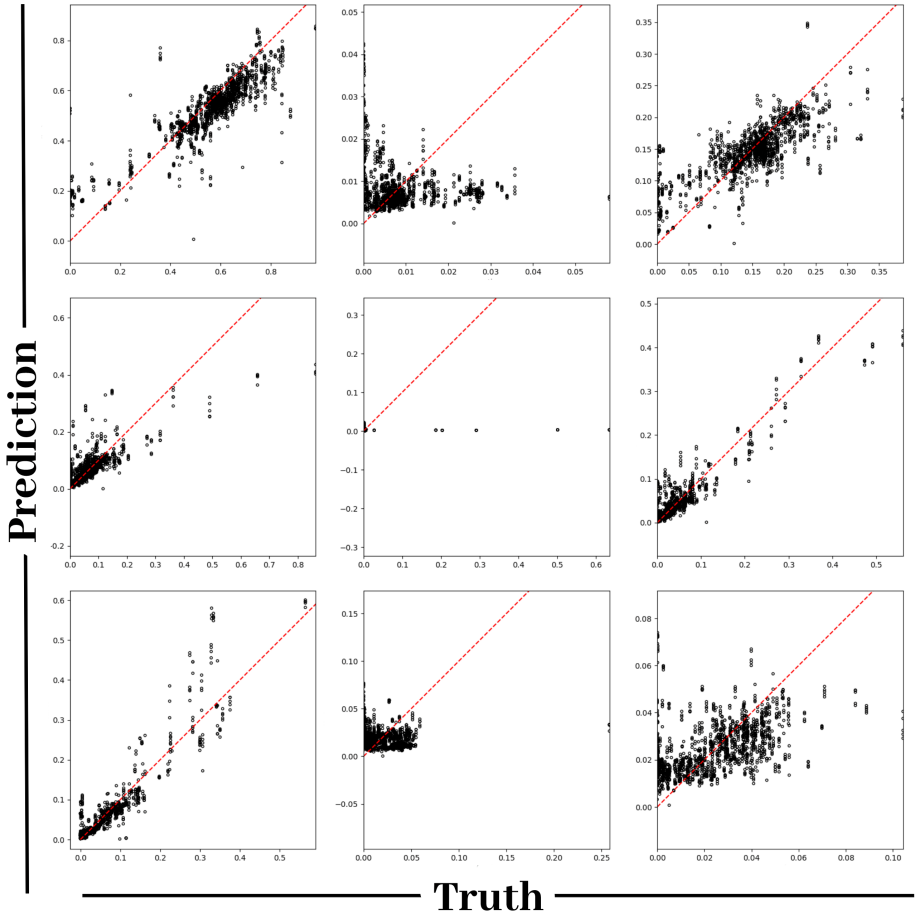


Illustration based on another illustration by:
https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.ht

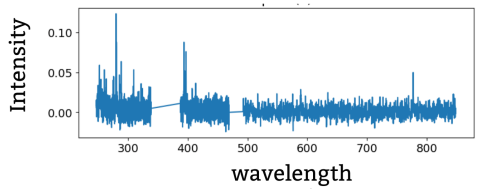
VAE results

Probabilistic Encoder CV Results

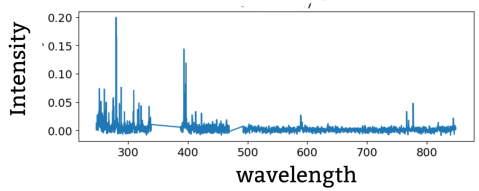
3-fold CV, 100 epochs
MAE : 2.155



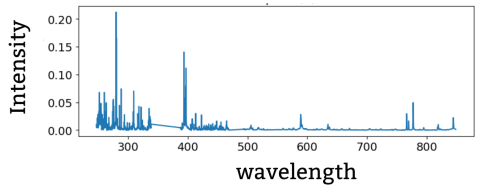
Probabilistic Decoder Results



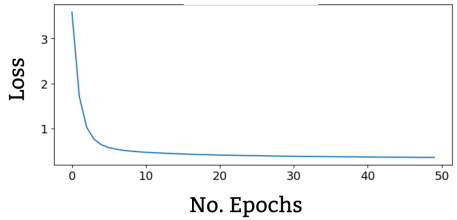
1 epoch

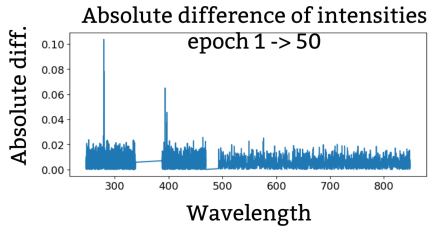


10 epochs

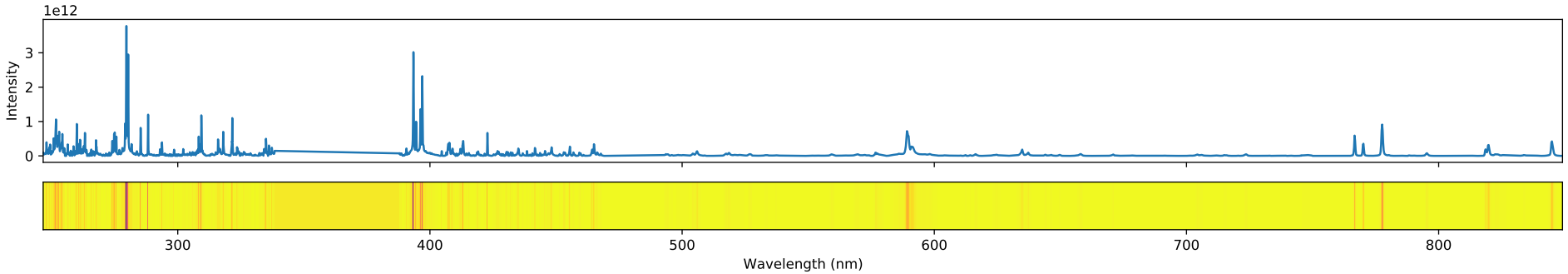


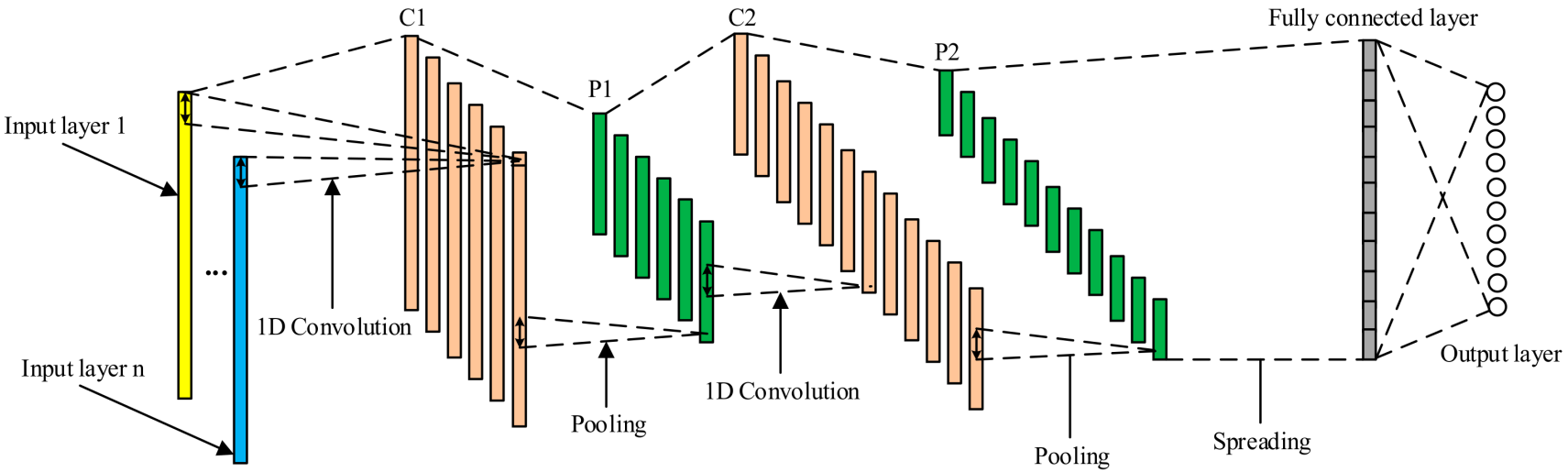
50 epochs



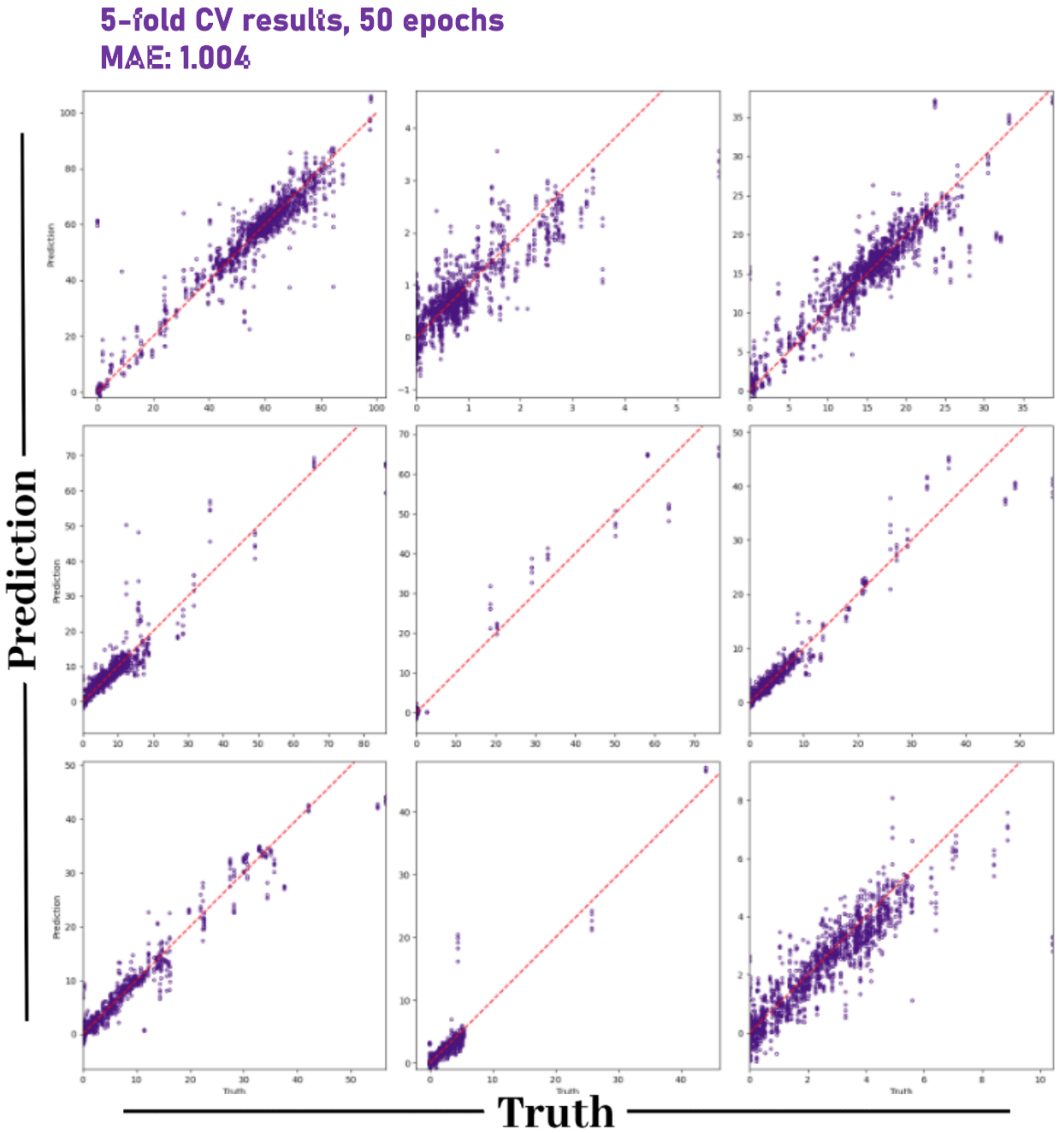


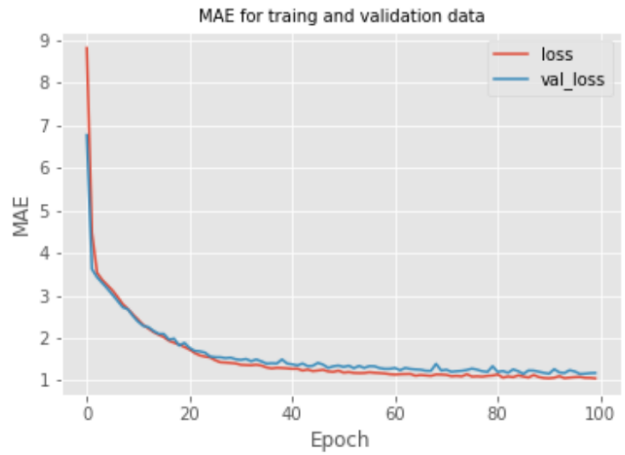
Convolutional NN model

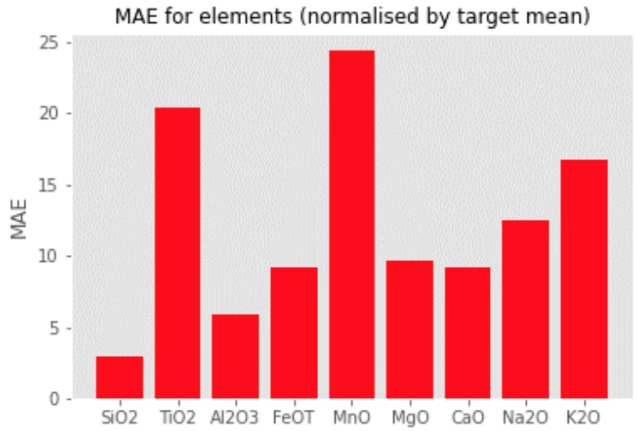




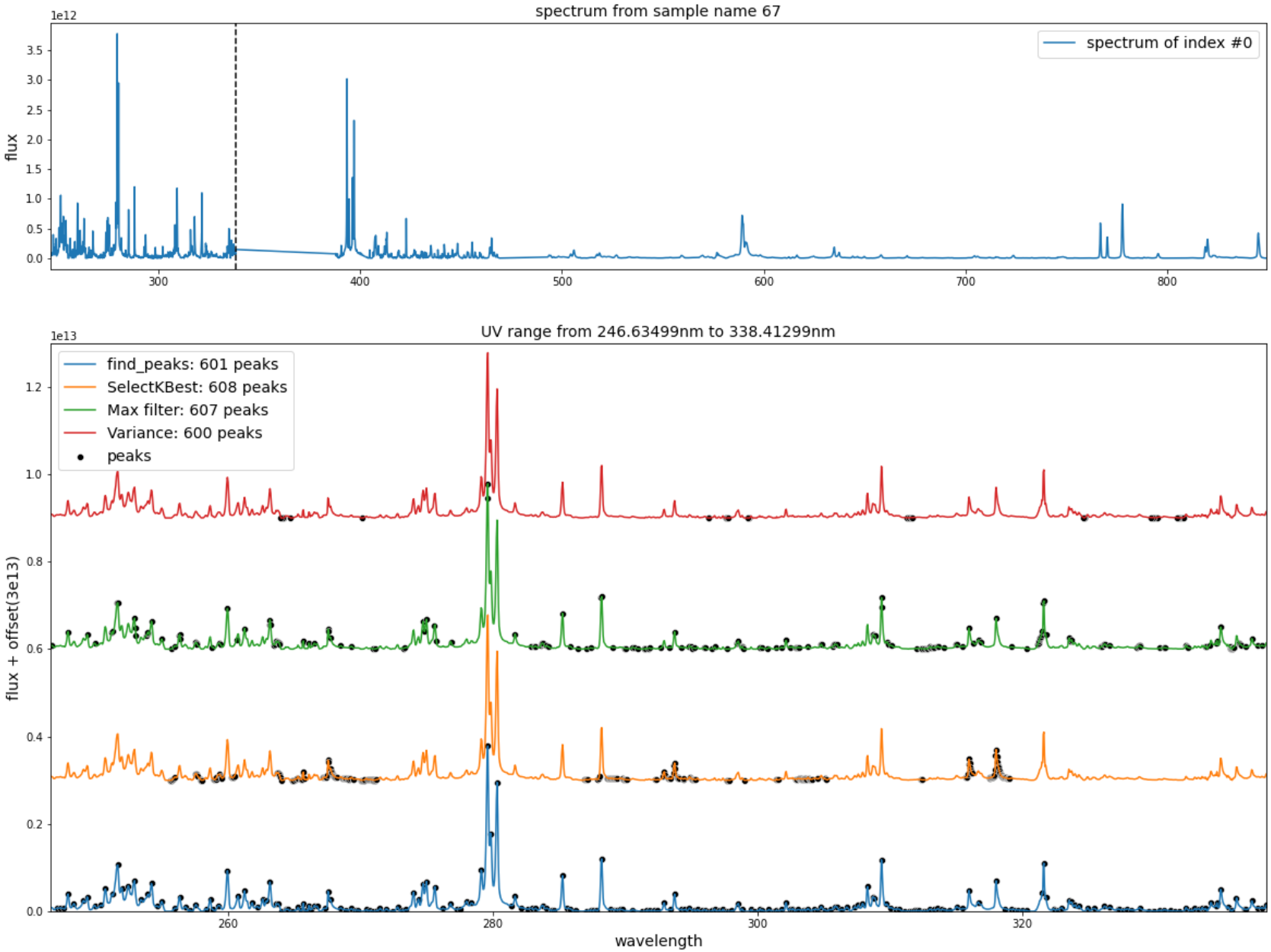
CNN Results


5-fold CV results, 50 epochs
MAE: 1.004
Prediction
Truth

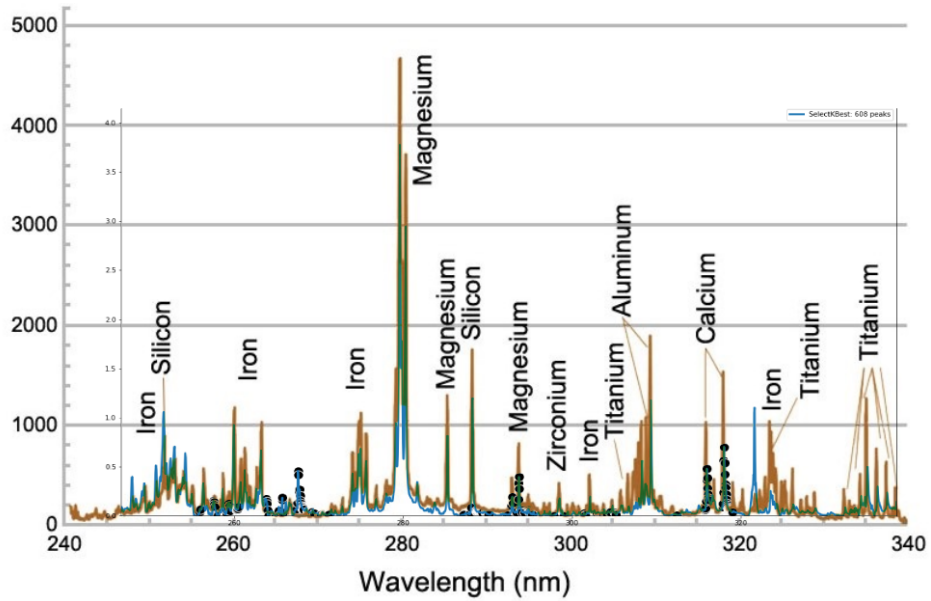

MAE for traing and validation data
loss
val_loss
MAE
Epoch


MAE for elements (normalised by target mean)
MAE
SiO2
TiO2
Al2O3
FeOT
MnO
MgO
CaO
Na2O
K2O

Model based on peak/feature selection



Peak/Feature model: Results



	SiO2	TiO2	Al2O3	FeOT	MnO	MgO	CaO	Na2O	K2O
0	58.400002	0.84	19.0	5.038928	0.07	2.96	3.1	5.26	2.16

	All data	find_peaks	SelectKbest	Max filter	Variance
peaks	5456.000	601.000	608.000	607.000	600.000
mae	0.955	0.942	0.938	0.929	1.235
std	0.155	0.156	0.108	0.110	0.126

- Alignment of 'SelectKBest' compared to NASA's element outline
- Concentration of the sample compared to spectrum
- performance on a simple tree based model

Conclusion and future work

Conclusion

- Ready to go to Mars? Not yet
- Promising models with good results
- LGBM performed the best, but we just scratched the surface of other more complex models

Some future prospects:

- More data!
- Expand on the VAE with either harsher reconstruction-loss or with convolutions
- Target-specific error analysis
- Optimise feature selection

"If you have only gone through the teaching process, even if you have been taught well, it doesn't become perfected until you go through the trial and error of using the principles you were taught. The action you take to prove the value of a principle is called experience"

Joan Jessalyn Cox

Honorable methods

- MLP
- XGBoost.Regressor

Appendix, Data cleaning(1/1)

We used the following methods to clean the data

- Removing outer wavelength layers that are not stable for the spectrometers used in ChemCam
- Removing samples with a total concentration above 100%
- Removing samples with different names but identical compositions

We did minor experiments with the following methods to clean the data, but they did not seem to improve the performance

- Removing features with less than n nonzero values
- Removing samples containing measurements more than n σ from the mean of a given feature
- Replacing samples containing measurements more than n σ from the mean of a given feature with the median of the remaining "clean" spectra from the same composition
- Replacing samples containing measurements more than n σ from the mean of a given feature with the mean of the remaining "clean" spectra from the same composition

Appendix, Scaling tests(1/2)

The testing of scaling methods was made using our spectral data (after cleaning) with scalers included in scikit-learn's sklearn.preprocessing package. The scalers were called with default parameters, apart from Normalizer, with which we tested each of its three norms. We then tested a sequence of normalising and standardisation, and vice versa.
They were tested with both a tree based- and a neural net based multiple output regressor. The tree based regressor was DecisionTreeRegressor from sklearn.tree, with default parameters. The neural network was a Keras multi layer perceptron model with the following configuration:

```
{'name': 'sequential_260', 'layers': [{'class_name': 'InputLayer', 'config': {'batch_input_shape': (None, 5477)
, 'dtype': 'float32', 'sparse': False, 'ragged': False, 'name': 'dense_520_input'}}, {'class_name': 'Dense', 'c
onfig': {'name': 'dense_520', 'trainable': True, 'batch_input_shape': (None, 5477), 'dtype': 'float32', 'units'
: 20, 'activation': 'relu', 'use_bias': True, 'kernel_initializer': {'class_name': 'HeUniform', 'config': {'see
d': None}}, 'bias_initializer': {'class_name': 'Zeros', 'config': {}}, 'kernel_regularizer': None, 'bias_regula
rizer': None, 'activity_regularizer': None, 'kernel_constraint': None, 'bias_constraint': None}}, {'class_name'
: 'Dense', 'config': {'name': 'dense_521', 'trainable': True, 'dtype': 'float32', 'units': 9, 'activation': 'li
near', 'use_bias': True, 'kernel_initializer': {'class_name': 'GlorotUniform', 'config': {'seed': None}}, 'bias
_initializer': {'class_name': 'Zeros', 'config': {}}, 'kernel_regularizer': None, 'bias_regularizer': None, 'ac
tivity_regularizer': None, 'kernel_constraint': None, 'bias_constraint': None}}]}
```

In both cases we used 10-fold cross validation and mean absolute error as the scoring metric. In the full scaling test we look at two ways of scaling the data: the first is the standard `X_scaled = scaler.fit_transform(X_unscaled)` which returns an array, and the second fills the columns of an output dataframe with scaled columns sequentially. These methods yield slightly different results, as seen in the full table.

Appendix, Scaling tests(2/2)

	Scaler	Tree MAE mean (array)	Tree MAE std (array)	Tree MAE mean (df)	Tree MAE std (df)	MLP MAE mean (df)	MLP MAE std (df)
0	StandardScaler	0.946222	0.151345	0.947396	0.151204	1.026088	0.048459
1	RobustScaler	0.946883	0.137617	0.956538	0.147564	3125.460846	5330.609618
2	MinMaxScaler	0.987789	0.147233	0.94273	0.137875	1.767521	0.350097
3	MaxAbsScaler	0.958454	0.155762	0.974464	0.13902	1.667965	0.280593
4	Normalizer	0.803676	0.123738	0.799355	0.127723	1.870137	0.121511
5	QuantileTransformer	0.971722	0.142552	0.964158	0.13902	1.523372	0.243001
6	PowerTransformer	0.93557	0.147046	0.935584	0.147196	1.094539	0.083177

Figure: Extended Scaling test table (default scaler params).

Additional results: Transform Normalizer(L2) then StandardScaler:

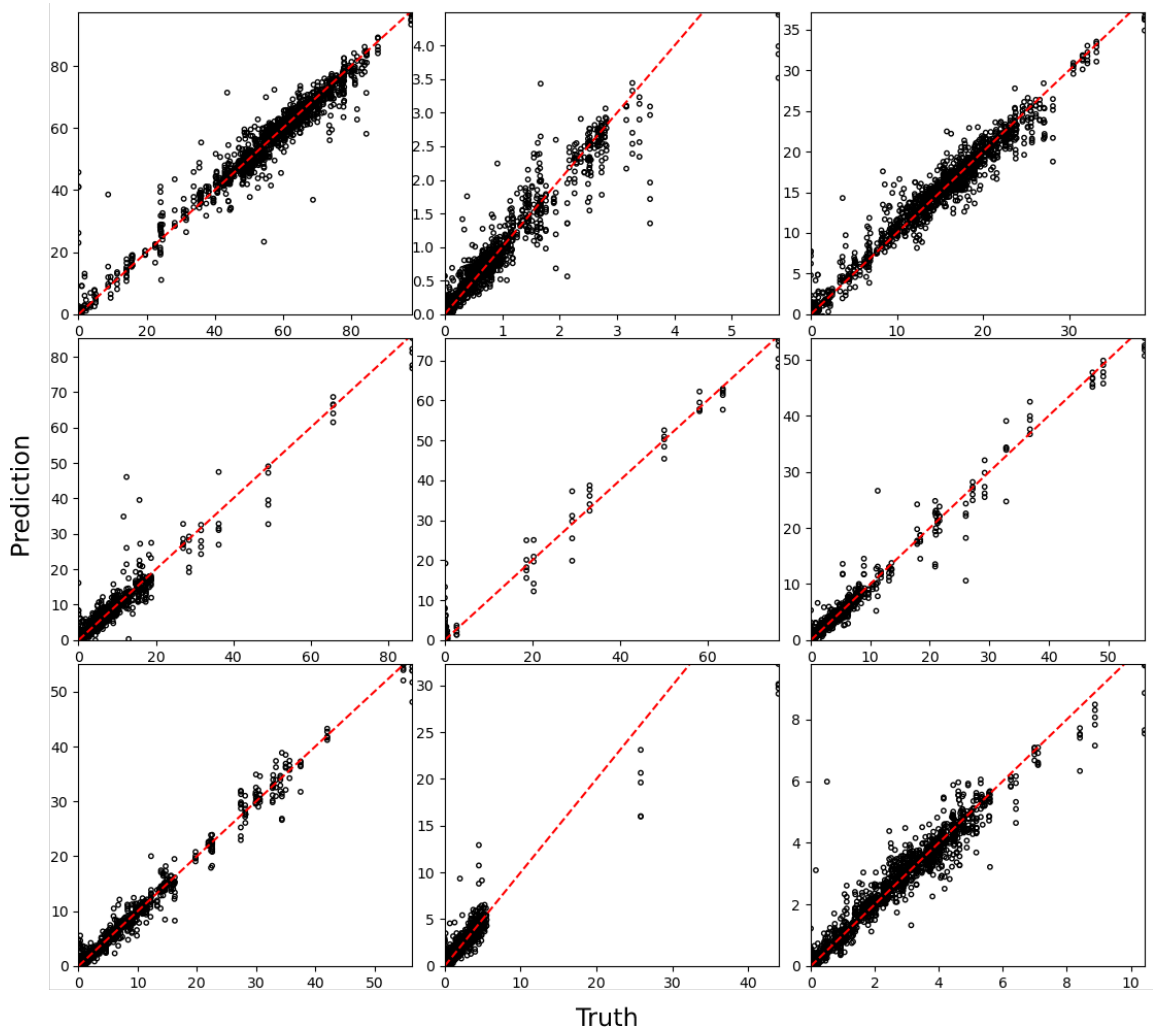
1. Tree MAE mean: 0.8035991842757276; Std: 0.11346258358688244
2. MLP MAE mean: 1.150329

Transform StandardScaler then Normalizer(L2):

1. Tree MAE mean: 0.9339504480171648; Std: 0.11944412657143666
2. MLP MAE mean: 1.1589469909667969

Appendix, LGBM (1/2) Prediction scatter-plot

Scatter plot of all predictions, done with 5-fold cross validation and default parameters in LGBMRegressor().



Appendix, LGBM (2/2) Optimized HP

As can be seen in the following table many of the targets found the same Hyper Parameters. We have no explanation for this, other than it might be some sort of local minima. Optimizing for all 9 targets took 5 hours on a 4-core PC.

	colsample_bytree	min_child_samples	min_child_weight	num_leaves	reg_alpha	subsample
SiO2	0.8184383024764295	42	0.1	48	2	0.6907023111377664
TiO2	0.7977397664700496	23	10	22	2	0.9637390248023816
Al2O3	0.7871685629560887	39	1	35	7	0.6796151700990114
FeOT	0.7977397664700496	23	10	22	2	0.9637390248023816
MnO	0.7977397664700496	23	10	22	2	0.9637390248023816
MgO	0.7977397664700496	23	10	22	2	0.9637390248023816
CaO	0.7977397664700496	23	10	22	2	0.9637390248023816
Na2O	0.7977397664700496	23	10	22	2	0.9637390248023816
K2O	0.7977397664700496	23	10	22	2	0.9637390248023816

Appendix, VAE (1/3) (Pros/Cons)

Potential Pros:

- The VAE is probabilistic, meaning we can potentially uncover the *secrets* of the latent space (composition space), and produce new "unseen" data - aka. generative sampling.
- The *stocasticity* is abstracted into the epsilon variable, which allows for efficiently making the VAE probabilistic.
- The latent space can be regularised using the KL-divergence loss.
  - The KL divergence loss term is trying to center the latent distributions and unifying the standard deviations, strengthening the connection between the distributions of each latent component, to allow for potential exploration of the latent space in generative sampling.
- The prior knowledge of latent *means* can be used to train the formation of the latent space.
- One could monitor the reconstruction loss and look for anomalies (spikes in the loss) in test-data.
- If given enough training data and confident latent means, one could use the VAE for de-noising testing-data by running it through the entire autoencoder.

Potential Cons:

- The VAE is difficult to train for high dimensional data - which applies to the LIBS data.
- Collapsing the latent space to Gaussian approximations may not be representative of the underlying distribution.
- As with PCA and other *lossy* dimensionality reduction techniques, the VAE inherit the *curse of dimensionality*; The higher the dimensionality of the problem the more samples are needed to estimate the reduced function.

Appendix, VAE (2/3) (Loss functions)

Experimental VAE that uses a MAE reconstruction loss and a additional regression loss term that tries to minimise the mean of the latent space to that of the input targets (and also minimise the variance).

σ : *Log SD of latent space*

μ : *Mean of latent space*

X_i^{in} : *Input features*

X_i^{out} : *Output features*

y_i^{in} : *Input targets*

$$\mathcal{L}_{recon} = \frac{\sum_{i=1}^{N} \left| X_i^{in} - X_i^{out} \right|}{N} \tag{1}$$

$$\mathcal{L}_{KL} = \frac{1}{2} \sum_i 1 + \sigma_i - \mu_i^2 - \exp(\sigma_i) \tag{2}$$

$$\mathcal{L}_{reg} = \sum_i \left| \frac{\mu_i - y_i^{in}}{\exp(\sigma_i)} \right| + \frac{1}{2} \sum_i |\sigma_i| \tag{3}$$

Appendix, VAE, (3/3) (Model params)

Summary of encoder and decoder and the combined VAE. Compiled with the "adam" optimiser, trained on batch sizes of 72.

Model: "Encoder"

Layer (type)	Output Shape	Param #	Connected to
Encoder_input (InputLayer)	(None, 5477)	0	
dropout_1 (Dropout)	(None, 5477)	0	Encoder_input[0][0]
Encoder_inter1 (LeakyReLU)	(None, 5477)	0	dropout_1[0][0]
Encoder_inter2 (LeakyReLU)	(None, 5477)	0	Encoder_inter1[0][0]
Sample_z_mean (Dense)	(None, 9)	49302	Encoder_inter2[0][0]
Sample_z_log_var (Dense)	(None, 9)	49302	Encoder_inter2[0][0]
Sample_z (Lambda)	(None, 9)	0	Sample_z_mean[0][0] Sample_z_log_var[0][0]

Total params: 98,604
Trainable params: 98,604
Non-trainable params: 0

Model: "Decoder"

Layer (type)	Output Shape	Param #
Sample_z_Input (InputLayer)	(None, 9)	0
Decoder_inter1 (Dense)	(None, 36)	360
Decoder_inter2 (Dense)	(None, 550)	20350
Decoder_output (Dense)	(None, 5477)	3017827

Total params: 3,038,537
Trainable params: 3,038,537
Non-trainable params: 0

Model: "VAE"

Layer (type)	Output Shape	Param #	Connected to
Encoder_input (InputLayer)	(None, 5477)	0	
Input_truth (InputLayer)	(None, 9)	0	
Encoder (Model)	[(None, 9), (None, 9	98604	Encoder_input[0][0] Input_truth[0][0]
Decoder (Model)	(None, 5477)	3038537	Encoder[1][2]

Total params: 3,137,141
Trainable params: 3,137,141
Non-trainable params: 0

The model was trained on a ERDA DAG instance that have access to 8 compute threads/cores and 16GB of memory.

Appendix, CNN (1/2) (Pros/Cons)

Potential Pros:

- The CNN uses the entire spectrum, and moreover there is no need for manual feature selection. This means that the modeller does not require great expertise in the field to manually input features of importance.
- Convolutions preserve the relationship between successive wavelengths ("pixels") and in fact this is encouraged for peaks/troughs in the data.
- Utilising many filters in the convolutional layers can allow many different patterns and details to be identified. Early layers pick out more low-level features before later layers recognise more high-level features.
- The CNN model could be very easily extended. Trivial to include more elements in the output. One could also include a custom loss function, constraining output to sum to 1 or penalising model complexity.

Potential Cons:

- Requires normalisation of input features, potentially leading to a loss of information.
- The number of parameters in the model grows rapidly with the number of layers. This bears a heavy computational cost in terms of time to train the model. This is exacerbated for HPO (using keras-tuner module).
- CNNs tend to require a lot of training data, perhaps more than we had available. Without this the model is susceptible to under-fitting.

Appendix, CNN (2/2) (Model Architecture)

The model summary for the CNN is shown below and included in the code supplied with our presentation. The CNN was trained solely on my local cpu, training took approximately 1 hour without parallelization.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_4 (Conv1D)            (None, 5473, 25)          150
_________________________________________________________________
max_pooling1d_4 (MaxPooling1 (None, 2736, 25)          0
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 2732, 64)          8064
_________________________________________________________________
max_pooling1d_5 (MaxPooling1 (None, 683, 64)           0
_________________________________________________________________
flatten_2 (Flatten)          (None, 43712)             0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                2797632
_________________________________________________________________
dense_7 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_8 (Dense)              (None, 9)                 297
=================================================================
Total params: 2,808,223
Trainable params: 2,808,223
Non-trainable params: 0
_________________________________________________________________
```

Appendix, MLP(1/3),(Description)

This model was made at the very end of the project, which is the reason it is not included in the presentation. It seems to have a great potential and shows really good results without any major optimisation done.

Method:

- Create MLP for each target, with target-specific inputs, then concatenates all weights before producing output
- Feature selection for each target: The N features with greatest linear correlation to the target, separated by at least a distance L ($|\lambda_2 - \lambda_1| > L$), in order to avoid using features that are "too" correlated

Potential Pros:

- Very simple to build using Keras *Functional API*
- Fast to train
- Target specific optimisation, instead of all at once
- Features are used independently in between MLPs, allowing to use features in different ways for different models

Potential Cons:

- At first sight it performs worse than LGBM
- The amount of parameters may increase rapidly when using more input data, which possibly requires more neurons per hidden layer

Appendix, MLP(2/3) (Model params)

Summary of model

- 1 input layer with 100 entries for each target
- 2 dense hidden layers for each target
  1. 200 neurons
  2. 100 neurons
- 1 concatenate layer
- 1 output layer with 9 outputs
- Optimizer: "Adam" with exponential scheduling
- Loss: Mean absolute error (MAE)

Model: "model"

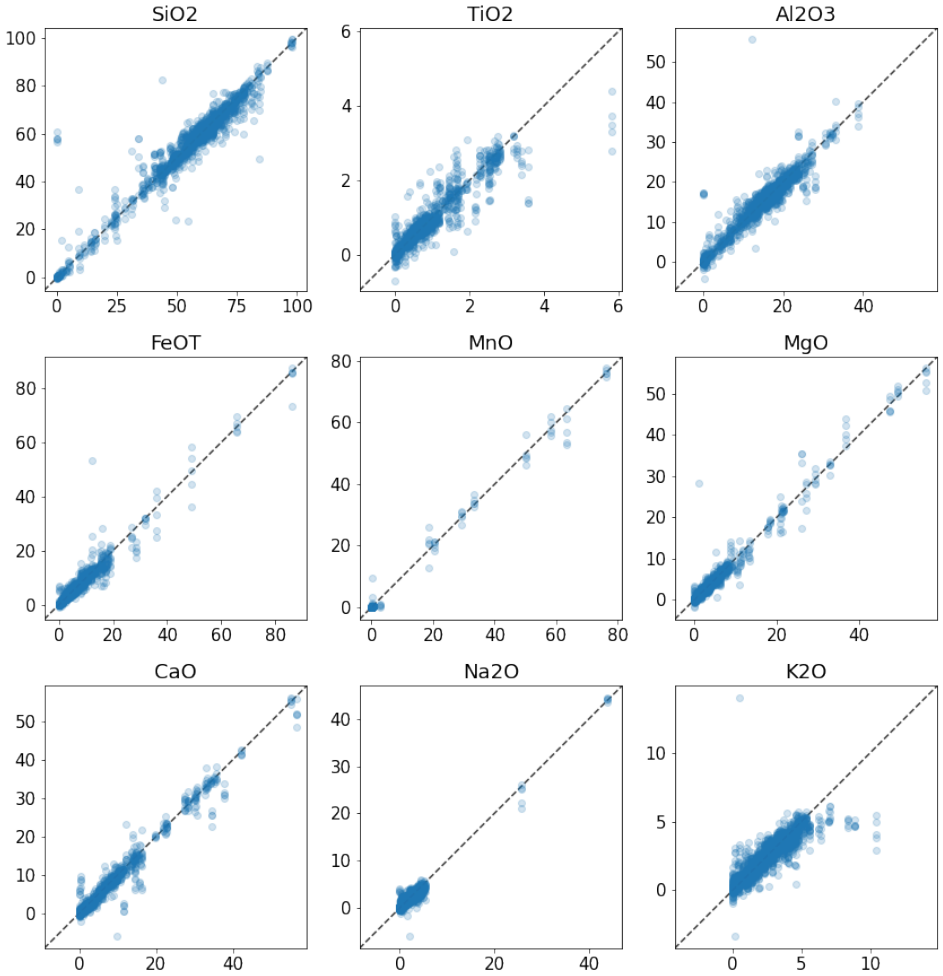
Layer (type)	Output Shape	Param #	Connected to
SiO2_input (InputLayer)	[(None, 100)]	0	
TiO2_input (InputLayer)	[(None, 100)]	0	
Al2O3_input (InputLayer)	[(None, 100)]	0	
FeOT_input (InputLayer)	[(None, 100)]	0	
MnO_input (InputLayer)	[(None, 100)]	0	
MgO_input (InputLayer)	[(None, 100)]	0	
CaO_input (InputLayer)	[(None, 100)]	0	
Na2O_input (InputLayer)	[(None, 100)]	0	
K2O_input (InputLayer)	[(None, 100)]	0	
dense (Dense)	(None, 200)	20200	SiO2_input[0][0]
dense_2 (Dense)	(None, 200)	20200	TiO2_input[0][0]
dense_4 (Dense)	(None, 200)	20200	Al2O3_input[0][0]
dense_6 (Dense)	(None, 200)	20200	FeOT_input[0][0]
dense_8 (Dense)	(None, 200)	20200	MnO_input[0][0]
dense_10 (Dense)	(None, 200)	20200	MgO_input[0][0]
dense_12 (Dense)	(None, 200)	20200	CaO_input[0][0]
dense_14 (Dense)	(None, 200)	20200	Na2O_input[0][0]
dense_16 (Dense)	(None, 200)	20200	K2O_input[0][0]
dense_1 (Dense)	(None, 100)	20100	dense[0][0]
dense_3 (Dense)	(None, 100)	20100	dense_2[0][0]
dense_5 (Dense)	(None, 100)	20100	dense_4[0][0]
dense_7 (Dense)	(None, 100)	20100	dense_6[0][0]
dense_9 (Dense)	(None, 100)	20100	dense_8[0][0]
dense_11 (Dense)	(None, 100)	20100	dense_10[0][0]
dense_13 (Dense)	(None, 100)	20100	dense_12[0][0]
dense_15 (Dense)	(None, 100)	20100	dense_14[0][0]
dense_17 (Dense)	(None, 100)	20100	dense_16[0][0]
concatenate (Concatenate)	(None, 900)	0	dense_1[0][0] dense_3[0][0] dense_5[0][0] dense_7[0][0] dense_9[0][0] dense_11[0][0] dense_13[0][0] dense_15[0][0] dense_17[0][0]
output (Dense)	(None, 9)	8109	concatenate[0][0]

Total params: 370,809
Trainable params: 370,809
Non-trainable params: 0

Appendix, MLP(3/3),(Results)

Result for 6-fold CV with $N = 100$ features per target, with at least $L = 1$ nm inbetween each feature. The model was trained solely on my local CPU, 6-fold CV took approximately 15 minutes.

Target	**MAE**
All targets	0.77 ± 0.05
SiO_2	2.4 ± 0.3
TiO_2	0.15 ± 0.01
Al_2O_3	1.21 ± 0.08
$FeOT$	0.90 ± 0.06
MnO	0.11 ± 0.03
MgO	0.50 ± 0.07
CaO	0.67 ± 0.08
Na_2O	0.47 ± 0.03
K_2O	0.51 ± 0.04



XGBoost results and hyperparameter optimization

Our XGBoost model development ran in parallel with LGBM until LGBM showed both faster performance and better results.

Pros:

- Faster than LGBM when used on GPU

Cons:

- Slower than LGBM when on CPU
- overall through the course has had a lower accuraccy compared to LGBM

Hyperparameter optimization done using RandomSearchCV with 100 iterations and 3 fold cross validation. for the validation set achieved a mean absolute error (MAE) of 0.75111. the Lightgbm and Xgboost.regressor model used the same variables for optimization but was beaten by Lightgbm.

hyperparameter optimization values found using RandomSearchCV					
max_depth	21	learning_rate	0.021	min_child_weight	0.001
subsamples	0.9509617	reg_alpha	0.1	min_child_samples	212
colsample_bytree	1				

Executed on a system with 16Gb memory and an i7-6650U CPU and ran for a few hours with optimization.