Data Analysis Assignment 1

Cameron Smith

June 12, 2021

1 Question 1

1.1 Task 1

We want to determine the decay rate α for the radioactive material. A plot of the data is shown in Figure 1. To find α, we use a Stan model with parameters

$$\alpha \sim \mathcal{U}\left(0, \alpha_{max}\right) ,$$
$$N_{0,i} \sim \mathcal{U}\left(0, N_{0,max}\right)$$

and

$$t_{0,i} \sim \mathcal{U}\left(t_{0,min}, t_{0,max}\right) ,$$

and model

$$N_{1,i} \sim \mathcal{N}\left(N_{0,i} \exp\left(-\alpha\left[t_{i,1} - t_{i,0}\right]\right), \sigma_i\right) ,$$

where $N_{0,i}$ is the material in the widget i at the time of manufacture, $N_{1,i}$ is the material in the widget i at the time of observation, $t_{1,i}$, $N_{0,max} = 20\,g$ is the maximum amount of material in a newly manufactured widget, $t_{0,min}$ and $t_{0,max}$ refer to the time at the beginning and end of manufacturing respectively, σ_i is the uncertainty in each measurement, and

$$\alpha_{max} = -\frac{1}{\Delta t_{\text{delay}}} \log\left(\frac{\min(N_1)}{\max(N_0)}\right)$$

is the maximum possible decay rate that could produce any of our data.

The model was optimised then sampled over. The sampling chains and resulting log posterior distribution is shown in Figure 2. Through sampling we found a value for α of

$$\alpha = (5.8 \pm 0.5) \times 10^{-8}\,\text{s}^{-1} .$$

1.2 Task 2

We now want to consider bias in one (unknown) detector. To account for this, we introduce new parameters b, the level of bias, and θ_k, the probability that detector k has a bias. We have

$$\theta_k \sim \text{Multinomial}(K)$$

and

$$b \sim \mathcal{U}\left(-\infty, \infty\right) .$$

The mean of each observed quantity by a given detector ($N_{\text{obs},k,i}$) is then related to the true quantity ($N_{\text{true},k,i}$) by

$$\mathbb{E}(N_{\text{obs},k,i}) = \mathbb{E}(N_{\text{true},k,i} + B_k),$$

where B_k is the random variable corresponding to the bias, given by the distribution:

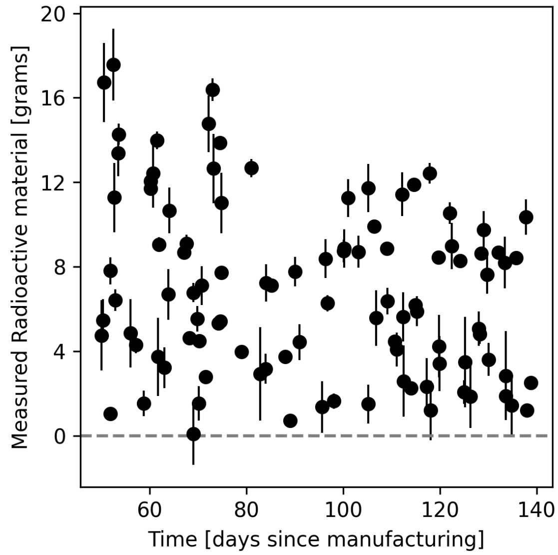


Figure 1: Radioactivity as a function of time

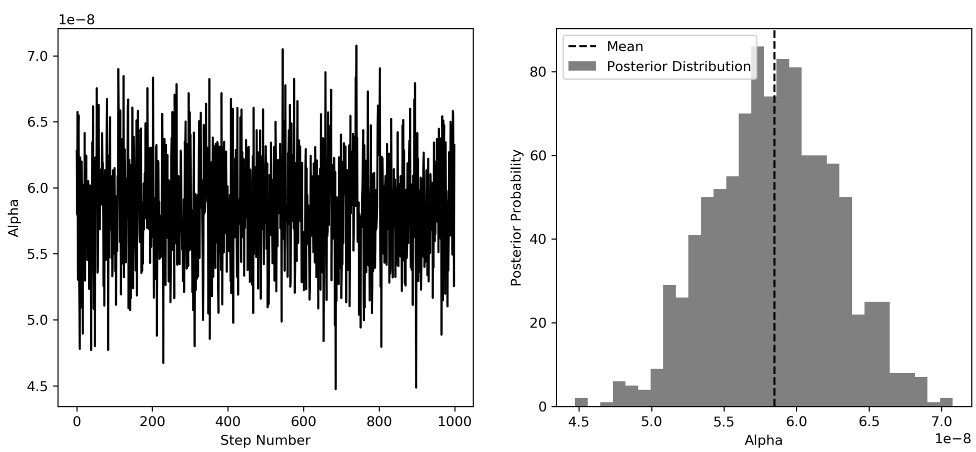


Figure 2: Sampling chains (left) and posterior probability (right) of the decay rate α.

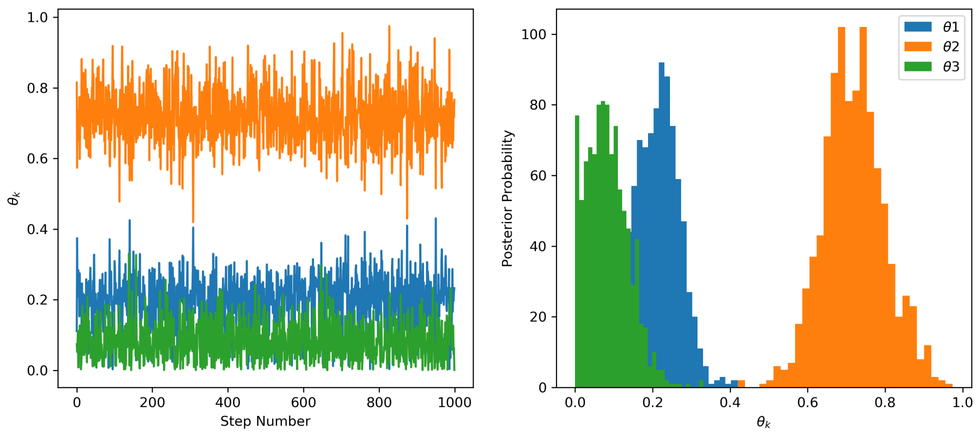


Figure 3: Sampling chains (left) and posterior probability (right) for the parameters θ_k. θ_1, θ_2 and θ_3 correspond to detectors A, B and C respectively.

B_k	b	0
$P(B_k)$	θ_k	$1-\theta_k$

We can therefore simplify our expression as follows

$$\begin{aligned}\mathbb{E}(N_{\mathrm{obs},k,i}) &= \mathbb{E}(N_{\mathrm{true},k,i}) + \mathbb{E}(B_k)\,, \\ &= N_{0,i}\exp\left(-\alpha\left[t_{i,1}-t_{i,0}\right]\right) + \theta_k b\,.\end{aligned}$$

Our model therefore becomes:

$$N_{1,k,i} \sim \mathcal{N}\left(N_{0,i}\exp\left(-\alpha\left[t_{i,1}-t_{i,0}\right]\right) + \theta_k b,\ \sigma_i\right)\,.$$

Our new model was again implemented in Stan, optimised and sampled over. Figures 3, 4 and 5 show the sampling chains and resulting posterior distributions of parameters θ, b and α respectively. From Figure 3 we can see that detector B is likely to be biased. The posterior distribution shown in Figure 4 gives the level of bias as:

$$\alpha = (4.2 \pm 0.8)\,\mathrm{g}\,.$$

Now that we have accounted for the bias, we get a new value for α:

$$\alpha = (8.2 \pm 0.7) \times 10^{-8}\,\mathrm{s}^{-1}\,.$$

The code for Question 1 is shown in Appendix A.

2 Question 2

2.1 Task 1

Figure 6(a) shows the brightness of a star as a function of time. Due to stellar rotation, the signal is quasi-periodic. Figure 6(b) shows a Lomb-Scargle periodogram of the data. The peaks with a power greater than 100 are:

Frequency [day^{-1}]	Power	Period [days]
0.123	303	51
0.229	1672	27
0.327	161	19
0.421	922	14
0.605	134	10

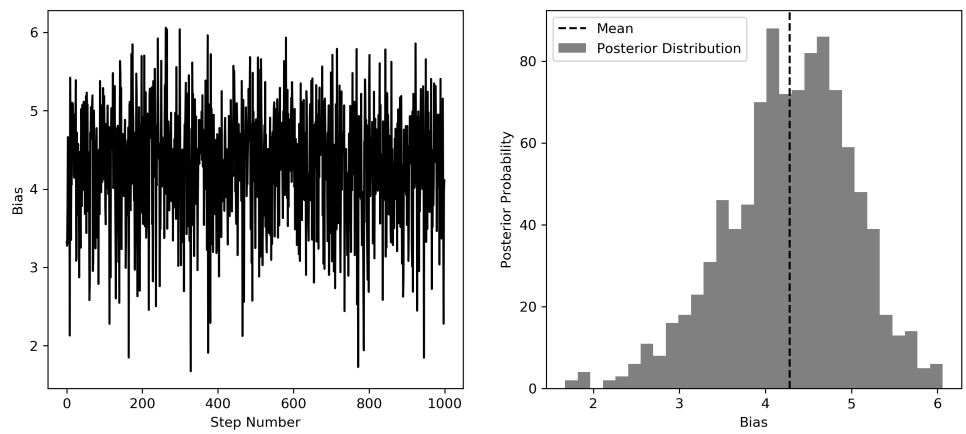


Figure 4: Sampling chains (left) and posterior probability (right) for the bias b.

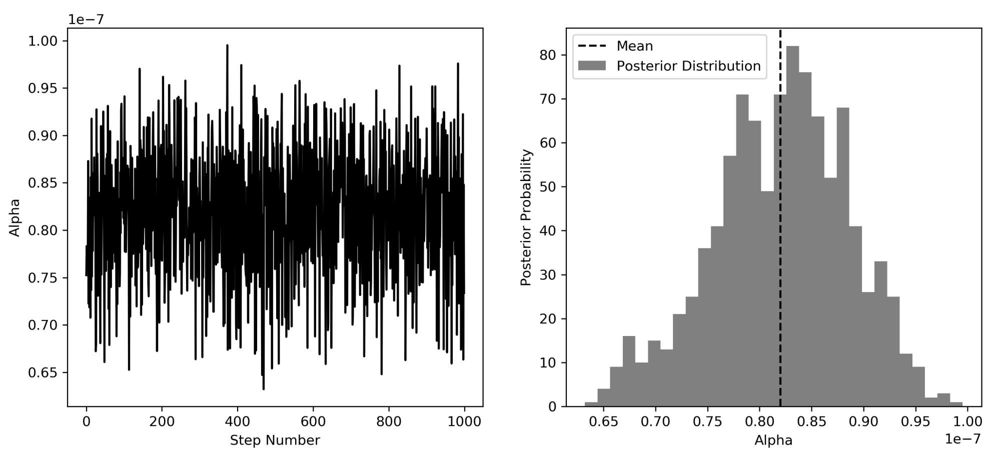
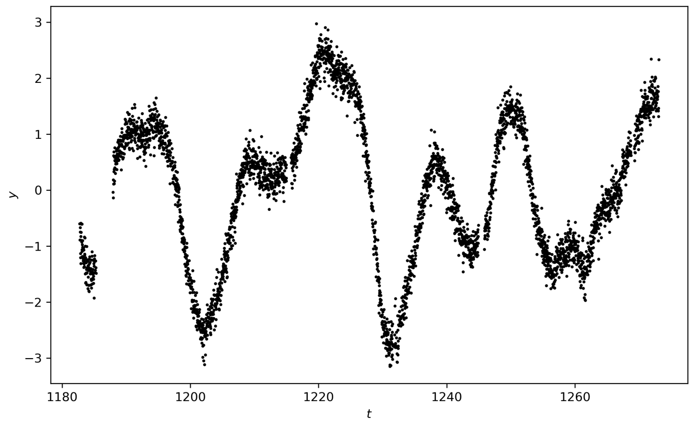
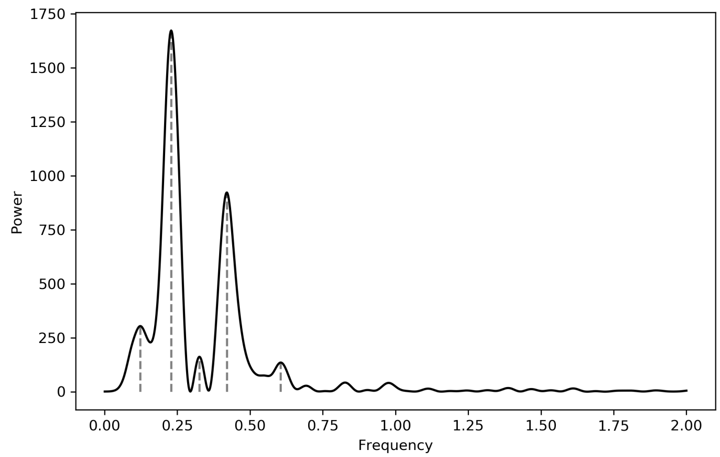


Figure 5: Sampling chains (left) and posterior probability (right) for the decay rate α, after accounting for a bias.



(a)



(b)

Figure 6: (a) The brightness of a star as a function of time. Error bars have been omitted for clarity. (b) The Lomb-Scargle periodogram of the data. Peaks with a power greater than 100 are indicated with vertical lines.

where the periods were calculated according to

$$T = \frac{2\pi}{f}\,,$$

where f is the frequency. The periodogram is dominated by the frequencies at $0.229\,\mathrm{day}^{-1}$ and $0.421\,\mathrm{day}^{-1}$, which possibly correspond to different stellar rotation rates at different latitudes.

2.2 Task 2

Now that we know the periodicity of the data, we can fit the data and make predictions of the future brightness using a gaussian process. To do this we used the following combination of kernals

$$k(\mathbf{x}_i,\, \mathbf{x}_j) = k_1(\mathbf{x}_i,\, \mathbf{x}_j)\,(k_2(\mathbf{x}_i,\, \mathbf{x}_j) + k_3(\mathbf{x}_i,\, \mathbf{x}_j))$$

where,

$$k_1(\mathbf{x}_i,\, \mathbf{x}_j) = \exp\left(-\frac{(x-x')^2}{2}\right)\,,$$
$$k_2(\mathbf{x}_i,\, \mathbf{x}_j) = \exp\left(-\Gamma_2\, \sin^2\left[\frac{\pi}{P_2}\,|x_i - x_j|\right]\right)\,, \text{ and}$$
$$k_3(\mathbf{x}_i,\, \mathbf{x}_j) = \exp\left(-\Gamma_3\, \sin^2\left[\frac{\pi}{P_3}\,|x_i - x_j|\right]\right)\,.$$

The periods P_1 and P_2 where set to match the data periodicity found in Task 1, with

$$P_2 = 0.229\,\mathrm{day}^{-1}\,, \text{ and}$$
$$P_3 = 0.421\,\mathrm{day}^{-1}\,,$$

while the scale of the correlations where initially set to

$$\Gamma_2 = \Gamma_3 = 1\,.$$

The Gaussian process was computed before optimising the parameters. Figure 7(a) shows predictions for the future brightness of this star for two months after the last data point. Figure 7(b) shows the same prediction after optimising over the parameters P_2, P_3, Γ_2 and Γ_3. Notably, P_2 and P_3 where *not* kept constant during optimisation, as the Lomb-Scargle periodogram (which was used to calculate their initial values) did not take into account the y-error. As can be seen in from Figures 7(a) and 7(b), the optimisation did not significantly change the predictions, however the predictions look sensible nonetheless. The code for Question 2 is shown in Appendix B.

3 Question 3

3.1 Task 1

In this task, we build and train an autoencoder from images of hand written digits. Our autoencoder was a fully connected dense neural network with 5 layers, starting with the 28×28 input, 3 hidden layers, and a 28×28 output. The middle layer was an 8 dimensional latent space. Specifically our autoencoder model is:

$$\boldsymbol{x} = \mathrm{AE}(\boldsymbol{x})\,,$$

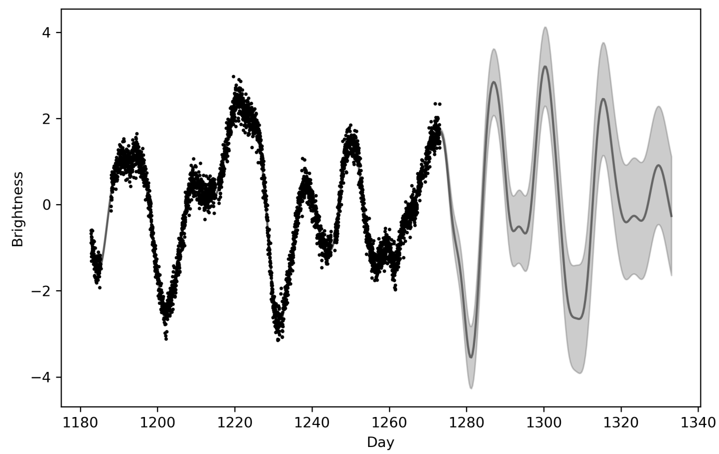
where,

$$\mathrm{AE}(\boldsymbol{x}) = \mathrm{OUT}(H_2(L(H_1(\boldsymbol{x}))))\,,$$

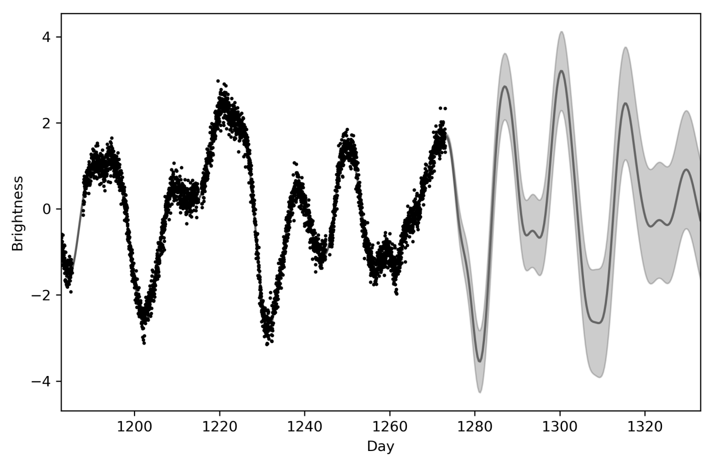
and,

$$H_1(\boldsymbol{x}) = \mathrm{ReLU}(W_{H_1}\boldsymbol{x} + \boldsymbol{b}_{H_1}\,,$$
$$L(\boldsymbol{x}) = \mathrm{ReLU}(W_L\boldsymbol{x} + \boldsymbol{b}_L\,,$$
$$H_2(\boldsymbol{x}) = \mathrm{Sigmoid}(W_{H_2}\boldsymbol{x} + \boldsymbol{b}_{H_2}\,,$$
$$\mathrm{OUT}(\boldsymbol{x}) = \mathrm{Sigmoid}(W_O\boldsymbol{x} + \boldsymbol{b}_O\,.$$

Weight matrix W_{H_1} has size (32×784), W_L has size (8×32), W_{H_2} has size (32×8), and W_O has size (784×32). Similarly, bias vector $\boldsymbol{b}_{H_1}$ is 32 dimensional, $\boldsymbol{b}_L$ is 8 dimensional, $\boldsymbol{b}_{H_2}$ is 32 dimensional, $\boldsymbol{b}_O$ is 784 dimensional. Training the network involves learning the elements of the aforementioned matrices and vectors, such that the output, $\mathrm{AE}(\boldsymbol{x})$, most closely matches $\boldsymbol{x}$. The autoencoder was trained for 50 epochs with a batch size of 256. Figure 8 shows some sample predictions of the autoencoder after training. The latent space (output of $L(H_1(\boldsymbol{x}))$) is shown in Figure 9.



(a)



(b)

Figure 7: (a) The Gaussian process predictions before optimisation. (b) The Gaussian process predictions after optimisation.

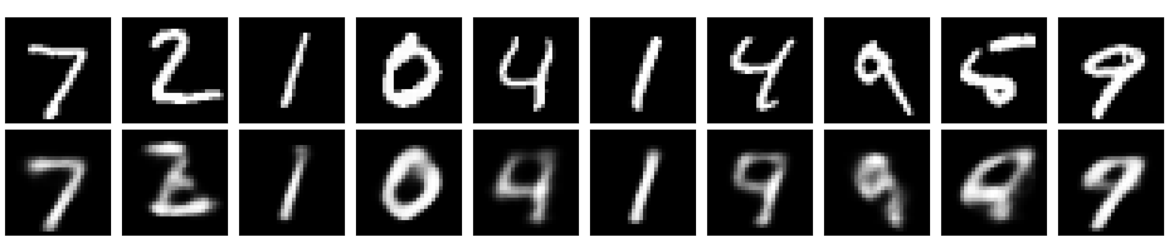


Figure 8: Sample inputs (top) and autoencoder predictions (bottom).

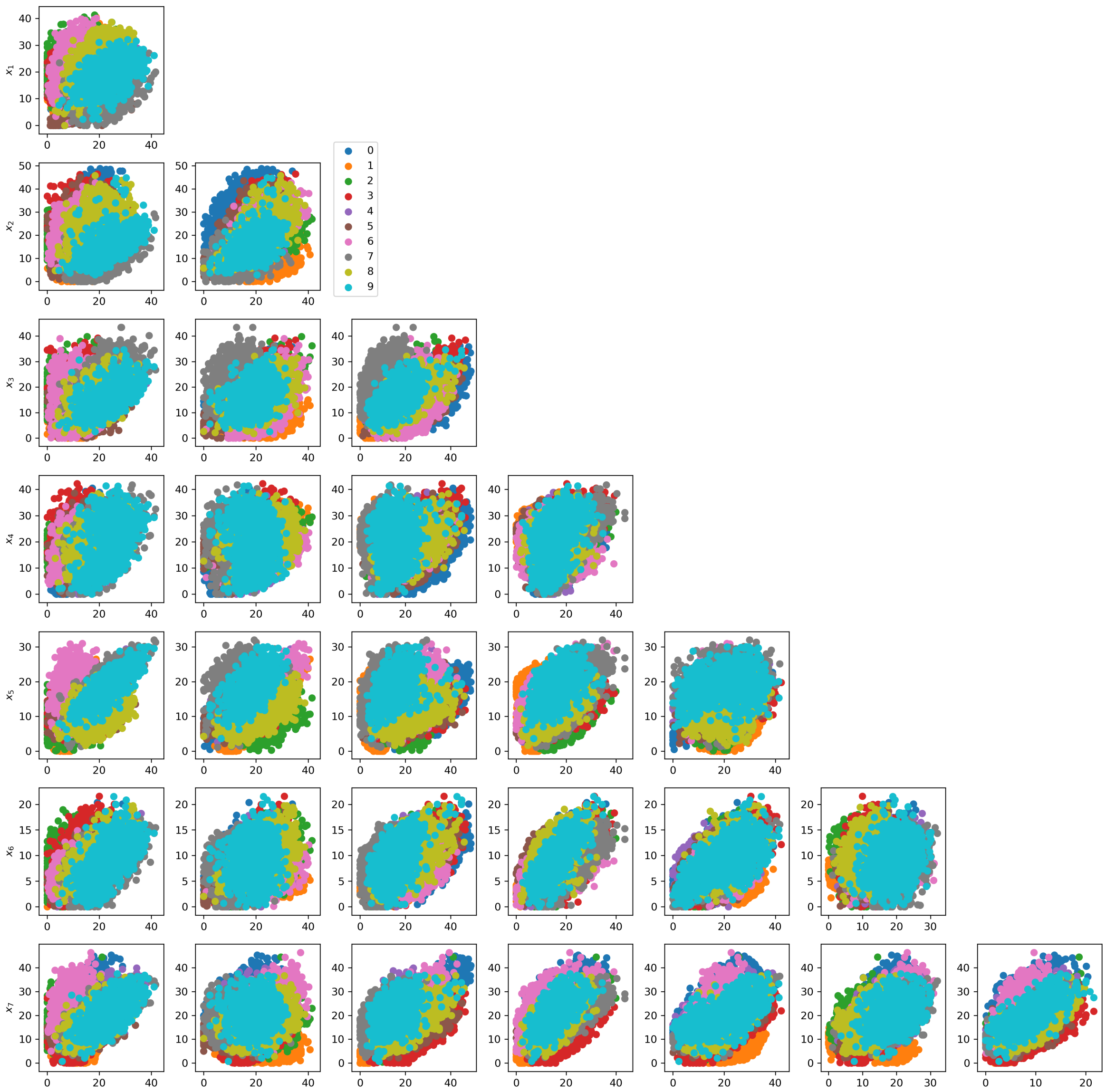


Figure 9: The autoencoder latent space.

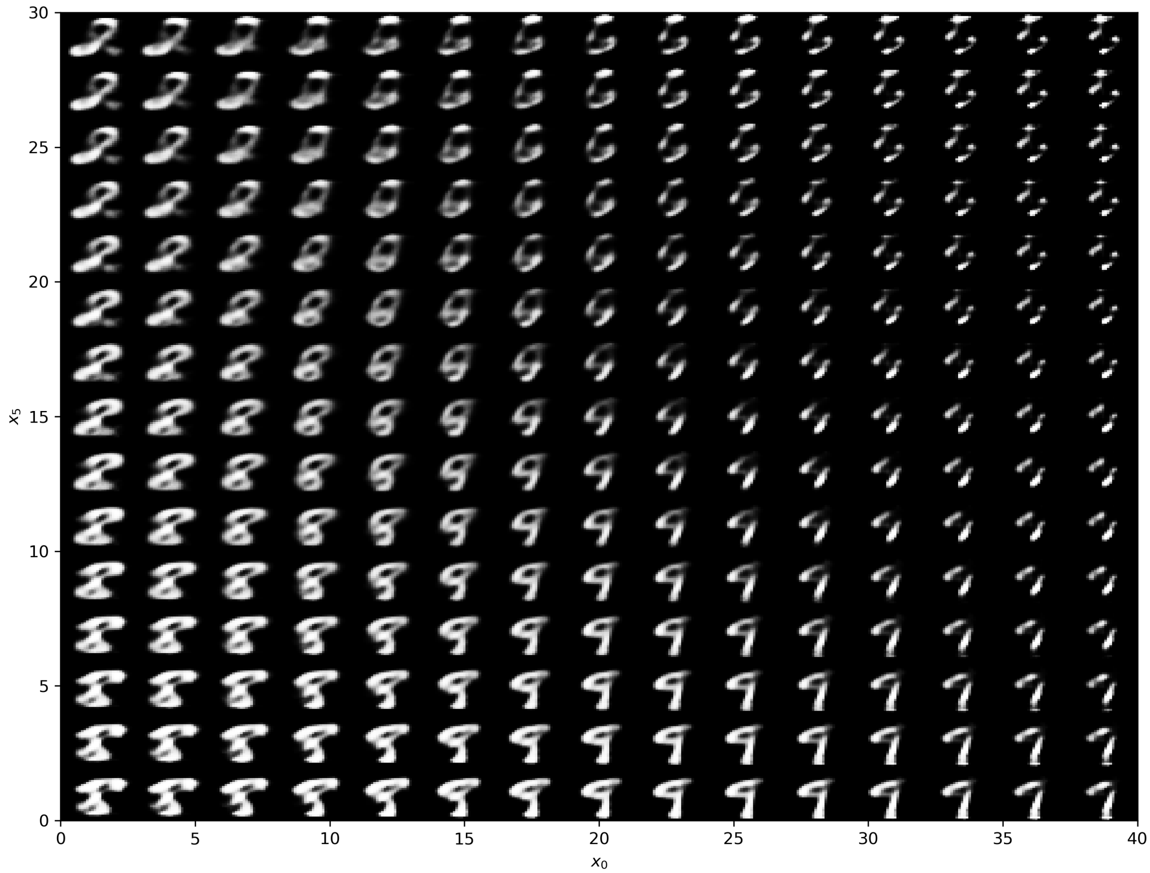


Figure 10: The autoencoder predictions at one slice of latent space.

3.2 Task 2

Figure 9 shows the autoencoder predictions across two dimensions of latent space. Since the latent space is 8 dimensional, the remaining 6 dimensions where kept constant. Specifically, we had

$$
\begin{aligned}
x_1 &= 20\,,\\
x_2 &= 20\,,\\
x_3 &= 10\,,\\
x_4 &= 20\,,\\
x_6 &= 20\,, \text{ and}\\
x_7 &= 20\,.
\end{aligned}
$$

while x_0 was plotted against x_5. These values were choosen so that the predictions were from latent space values near the data shown in Figure 10. The code for Question 3 is shown in Appendix C.

A Question 1

```python
# imports
import csv
import urllib.request as request
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import pystan as stan


# get data
url = "http://astrowizici.st/teaching/phs5000/decay.csv"
response = request.urlopen(url)
lines = [l.decode('utf-8') for l in response.readlines()]
cr = csv.reader(lines)
cr_data = list(cr)
data = np.array(cr_data[1:])
t, N, N_err, d_name = data.T
# time in seconds
t = t.astype(np.float)
# grams of radioactive material measured
N = N.astype(np.float)
# uncertainty in grams measured
N_err = N_err.astype(np.float)
# Detector name
d_name = d_name.astype(str)

# define useful quantities
days_to_s = 60 * 60 * 24
s_to_days = 1/days_to_s
# known parameters:
N_widgets = 100
N_initial_max = 20
manufacturing_time_span = 35 * days_to_s
t_delay = 14 * days_to_s
measurement_time_span = 90 * days_to_s
N_observations = 1
# time in days
t_days = t * s_to_days
# minimum measured N
min_N = np.min(N)
# Minimimum time span:
min_dt = t_delay
# maximum possible alpha value
max_alpha = -1/(min_dt) * np.log(min_N/N_initial_max)


# plot figure
fig, ax = plt.subplots(figsize=(4, 4))
ax.scatter(t_days, N, c="k", s=10)
ax.errorbar(t_days, N, yerr=N_err, fmt="o", lw=1, c="k")
ax.axhline(0, linestyle='--', c='gray')
ax.set_xlabel(r"Time [days since manufacturing]")
ax.set_ylabel(r"Measured Radioactive material [grams]")
ax.xaxis.set_major_locator(MaxNLocator(6))
ax.yaxis.set_major_locator(MaxNLocator(7))
fig.tight_layout()
plt.savefig('q1_data.png', dpi=300)


# pystan utils
def sampling_kwds(**kwargs):
    r"""
    Prepare a dictionary that can be passed to Stan at the sampling stage.
    Basically this just prepares the initial positions so that they match the
    number of chains.
    """

    kwds = dict(chains=4)
    kwds.update(kwargs)
```

```

    if "init" in kwds:
        kwds["init"] = [kwds["init"]] * kwds["chains"]

    return kwds


# define model
model_filename = "model1.stan"
model_str = """
data {
    int<lower=1> N_widgets; // number of widgets

    // Time of measurement.
    vector[N_widgets] t_measured;

    // Amount of material measured is uncertain.
    vector[N_widgets] N_measured;
    vector[N_widgets] sigma_N_measured;

    // Maximum amount of initial material.
    real N_initial_max;

    //  last possible time of manufacture.
    real t_initial_max;

    //  max value of alpha
    real alpha_max;
}

parameters {
    // Time of manufacture.
    vector<lower=0, upper=t_initial_max>[N_widgets] t_initial;

    // The decay rate parameter.
    real<lower=0, upper=alpha_max> alpha;

    // The amount of initial material is not known.
    vector<lower=0, upper=N_initial_max>[N_widgets] N_initial;
}

model {
    for (i in 1:N_widgets) {
        N_measured[i] ~ normal(
          N_initial[i] * exp(-alpha * (t_measured[i] - t_initial[i])),
          sigma_N_measured[i]
        );
    }
}
"""
# make model
model = stan.StanModel(model_code=model_str)

# Data.
data_dict = dict(
    N_widgets=N_widgets,
    t_measured=t,
    N_measured=N,
    sigma_N_measured=N_err,
    N_initial_max=N_initial_max,
    t_initial_max=manufacturing_time_span,
    alpha_max=max_alpha,
)

# initial guess
alpha_guess = max_alpha/2
t_init_guess = np.full(N_widgets, manufacturing_time_span/2)

init_dict = dict(
    t_initial=t_init_guess,
    alpha=alpha_guess,
```

```
)

# Run optimisation.
opt_stan = model.optimizing(
    data=data_dict,
    init=init_dict
)

# Run sampling.
samples = model.sampling(**sampling_kwds(
    chains=2,
    iter=2000,
    data=data_dict,
    init=opt_stan
))


# plot initial N
fig = samples.traceplot(("N_initial", ))
fig.set_size_inches(10, 5)
plt.savefig('q11_N_init.png', dpi=300)

# plot initial t
fig = samples.traceplot(("t_initial", ))
fig.set_size_inches(10, 5)
plt.savefig('q11_t_init.png', dpi=300)


# get alpha chain and remove burn in
alpha_chain = samples["alpha"][1000:]
# calculate mean and std
alpha_mean = np.mean(alpha_chain)
alpha_err = np.std(alpha_chain)

# plot
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

ax = axs[0]
ax.plot(alpha_chain, c='k')
ax.set_xlabel("Step Number")
ax.set_ylabel("Alpha")

ax = axs[1]
ax.hist(alpha_chain, bins=30, color='gray', label='Posterior Distribution')
ax.axvline(alpha_mean, c='k', linestyle='--', label='Mean')
ax.set_ylabel("Posterior Probability")
ax.set_xlabel("Alpha")
ax.legend()
plt.savefig('q11_alpha.png', dpi=300)


# print mean and std
print(f'Alpha: {alpha_mean:.2}, Standard Deviation: {alpha_err:.0}')


# Part 2


# make model
model_filename = "model2.stan"
model_str = """
data {
    int<lower=1> N_widgets; // number of widgets

    // Time of measurement.
    vector[N_widgets] t_measured;

    // Amount of material measured is uncertain.
    vector[N_widgets] N_measured;
    vector[N_widgets] sigma_N_measured;

```

```
    // Maximum amount of initial material.
    real N_initial_max;

    //  last possible time of manufacture.
    real t_initial_max;

    //  max value of alpha
    real alpha_max;

    //detector
    int k[N_widgets];
}

parameters {
    // Time of manufacture.
    vector<lower=0, upper=t_initial_max>[N_widgets] t_initial;

    // The decay rate parameter.
    real<lower=0, upper=alpha_max> alpha;

    // The detector bias.
    real bias;

    // The amount of initial material is not known.
    vector<lower=0, upper=N_initial_max>[N_widgets] N_initial;

    // multinomial
    simplex[3] theta;
}

model {
    for (i in 1:N_widgets) {
        N_measured[i] ~ normal(
          N_initial[i] * exp(-alpha * (t_measured[i] - t_initial[i]))
          + theta[k[i]] * bias,
          sigma_N_measured[i]
        );
    }
}
"""
model = stan.StanModel(model_code=model_str)


# convert detector names to numbers
name_to_k = dict(
    A=1,
    B=2,
    C=3,
)
k = np.vectorize(name_to_k.get)(d_name)

# Data.
data_dict = dict(
    N_widgets=N_widgets,
    t_measured=t,
    N_measured=N,
    sigma_N_measured=N_err,
    N_initial_max=N_initial_max,
    t_initial_max=manufacturing_time_span,
    alpha_max=max_alpha,
    k=k,
)

# initial guess
alpha_guess = max_alpha/2
t_init_guess = np.full(N_widgets, manufacturing_time_span/2)
theta_guess = [0.33, 0.33, 0.34]
bias_guess = 0.

init_dict = dict(
    t_initial=t_init_guess,
```

```python
    alpha=alpha_guess,
    theta=theta_guess,
    bias=bias_guess,
)

# Run optimisation.
opt_stan = model.optimizing(
    data=data_dict,
    init=init_dict
)

# Run sampling.
samples = model.sampling(**sampling_kwds(
    chains=2,
    iter=2000,
    data=data_dict,
    init=opt_stan,
    control=dict(max_treedepth=30),
))


# plot initial N
fig = samples.traceplot(("N_initial", ))
fig.set_size_inches(10, 5)
plt.savefig('q12_N_init.png', dpi=300)

# plot initial t
fig = samples.traceplot(("t_initial", ))
fig.set_size_inches(10, 5)
plt.savefig('q12_t_init.png', dpi=300)


# get theta chains and remove burn in
theta_chains = samples["theta"][1000:].T


# plot theta chains
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

ax = axs[0]
for i in range(3):
    ax.plot(theta_chains[i])
ax.set_xlabel("Step Number")
ax.set_ylabel(r"$\theta_k$")

ax = axs[1]
for i in range(3):
    ax.hist(theta_chains[i], bins=30, label=rf'$\theta${i+1}')
ax.legend()
ax.set_ylabel("Posterior Probability")
ax.set_xlabel(r"$\theta_k$")
plt.savefig('q12_theta.png', dpi=300)


# get bias chain and remove burn in
bias_chain = samples["bias"][1000:]
# calculate mean and std
bias_mean = np.mean(bias_chain)
bias_err = np.std(bias_chain)

# plot
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

ax = axs[0]
ax.plot(bias_chain, c='k')
ax.set_xlabel("Step Number")
ax.set_ylabel("Bias")

ax = axs[1]
ax.hist(bias_chain, bins=30, color='gray', label='Posterior Distribution')
ax.axvline(bias_mean, c='k', linestyle='--', label='Mean')
```

```
ax.set_ylabel("Posterior Probability")
ax.set_xlabel("Bias")
ax.legend()
plt.savefig('q12_bias.png', dpi=300)


# print mean and std
print(f'Bias: {bias_mean:.2}, Standard Deviation: {bias_err:.0}')


# get alpha chain and remove burn in
alpha_chain = samples["alpha"][1000:]
# calculate mean and std
alpha_mean = np.mean(alpha_chain)
alpha_err = np.std(alpha_chain)

# plot
fig, axs = plt.subplots(1, 2, figsize=(12, 5))

ax = axs[0]
ax.plot(alpha_chain, c='k')
ax.set_xlabel("Step Number")
ax.set_ylabel("Alpha")

ax = axs[1]
ax.hist(alpha_chain, bins=30, color='gray', label='Posterior Distribution')
ax.axvline(alpha_mean, c='k', linestyle='--', label='Mean')
ax.set_ylabel("Posterior Probability")
ax.set_xlabel("Alpha")
ax.legend()
plt.savefig('q12_alpha.png', dpi=300)


# print mean and std
print(f'Alpha: {alpha_mean:.2}, Standard Deviation: {alpha_err:.0}')
```

B Question 2

```
import scipy.optimize as op
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import pickle
from scipy.signal import lombscargle
from george import kernels
import george

# load data
with open("assignment2_gp.pkl", "rb") as fp:
    data = pickle.load(fp)
t = data['t']
y = data['y']
y_err = data['yerr']

# plot the data
fig, ax = plt.subplots(figsize=(8, 5))
ax.scatter(t, y, c="k", s=2)
ax.set_xlabel(r"$t$")
ax.set_ylabel(r"$y$")
ax.xaxis.set_major_locator(MaxNLocator(6))
ax.yaxis.set_major_locator(MaxNLocator(7))
fig.tight_layout()
plt.savefig('q2_data.png', dpi=300)


# get periodogram
f = np.linspace(0.001, 2, 1000)
pgram = lombscargle(t, y, f)


```

```python
# find index of peaks
peak_i = np.where(
    np.r_[True, pgram[1:] > pgram[:-1]] & np.r_[pgram[:-1] > pgram[1:], True]
)[0]
# power of peaks
peak_p = pgram[peak_i]

# exclude peaks with <100 power
peak_i = peak_i[np.where(peak_p > 100)]
peak_p = pgram[peak_i]
# frequency of peaks
peak_f = f[peak_i]

# plot
plt.figure(figsize=(8, 5))
plt.plot(f, pgram, c='k')
plt.xlabel("Frequency")
plt.ylabel("Power")
plt.vlines(peak_f, 0, peak_p, linestyle='--', colors='gray')
plt.savefig('periodogram.png', dpi=300)

# get periods (for later)
periods = 2*np.pi / peak_f
print('Frequency: \t Power: \t Periods:')
for freq, power, T in zip(peak_f, peak_p, periods):
    print(f'{freq:.3}\t \t {int(power)} \t \t {int(T)}')


# Part 2


# define our kernal
k1 = kernels.ExpSine2Kernel(gamma=1, log_period=np.log(periods[1]))
k2 = kernels.ExpSine2Kernel(gamma=1, log_period=np.log(periods[3]))

kernel = k1 + k2
kernel *= 1 * kernels.ExpSquaredKernel(1000)

# compute the gaussian process
gp = george.GP(kernel)
gp.compute(t, y_err)

t_min = np.min(t)
t_max = np.max(t)
t_pred = np.linspace(t_min, t_max + 60, 500)
pred, pred_var = gp.predict(y, t_pred, return_var=True)


plt.figure(figsize=(8, 5))
plt.fill_between(t_pred, pred - np.sqrt(pred_var), pred + np.sqrt(pred_var),
                 color="k", alpha=0.2)
plt.plot(t_pred, pred, "k", lw=1.5, alpha=0.5)
plt.scatter(t, y, c="k", s=2)
plt.xlim(t_min, t_max + 60)
plt.xlabel("Day")
plt.ylabel("Brightness")
plt.savefig('gp1.png', dpi=300)

print(f"Initial ln-likelihood: {gp.log_likelihood(y):.2f}")


# Define the objective function (negative log-likelihood in this case).
def nll(p):
    gp.set_parameter_vector(p)
    ll = gp.log_likelihood(y, quiet=True)
    return -ll if np.isfinite(ll) else 1e25


# And the gradient of the objective function.
def grad_nll(p):
    gp.set_parameter_vector(p)
```

```python
    return -gp.grad_log_likelihood(y, quiet=True)


# Run the optimization routine.
p0 = gp.get_parameter_vector()
results = op.minimize(nll, p0, jac=grad_nll, method="L-BFGS-B")

# Update the kernel and print the final log-likelihood.
gp.set_parameter_vector(results.x)
print(f"Final ln-likelihood: {gp.log_likelihood(y):.2f}")

t_min = np.min(t)
t_max = np.max(t)
t_pred = np.linspace(t_min, t_max + 60, 500)
mu, var = gp.predict(y, t_pred, return_var=True)


# plot
plt.figure(figsize=(8, 5))
plt.fill_between(t_pred, pred - np.sqrt(pred_var), pred + np.sqrt(pred_var),
                 color="k", alpha=0.2)
plt.plot(t_pred, pred, "k", lw=1.5, alpha=0.5)
plt.scatter(t, y, c="k", s=2)
plt.xlim(t_min, t_max + 60)
plt.xlabel("Day")
plt.ylabel("Brightness")
plt.savefig("gp2.png", dpi=300)
```

C Question 3

```python
import gzip
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow as tf


tf.random.set_seed(1)


# training/test set size
n_train = 60000
n_test = 10000


# get data function
def get_data(fname):
    image_size = 28
    f = gzip.open(fname, 'r')
    f.read(16)
    data = f.read()
    data = np.frombuffer(data, dtype=np.uint8).astype(np.float32)
    data = data.reshape(-1, image_size, image_size)
    f.close()
    return data


# get labels function
def get_labels(fname):
    f = gzip.open(fname, 'r')
    f.read(8)
    data = f.read()
    data = np.frombuffer(data, dtype=np.uint8)
    f.close()
    return data


# get data
train_data = get_data('train-images-idx3-ubyte.gz')
```

```
test_data = get_data('t10k-images-idx3-ubyte.gz')
labels = get_labels('t10k-labels-idx1-ubyte.gz')


# specify network
input_img = keras.Input(shape=(28*28,))

h_size = 32
latent_size = 8

h1 = layers.Dense(h_size, activation='relu')(input_img)
encoded = layers.Dense(latent_size, activation='relu')(h1)

h2 = layers.Dense(h_size, activation='relu')(encoded)
decoded = layers.Dense(28*28, activation='sigmoid')(h2)

autoencoder = keras.Model(input_img, decoded)

# just the encoder part
encoder = keras.Model(input_img, encoded)

# just the decoder part
encoded_input = keras.Input(shape=(latent_size,))
h2_layer = autoencoder.layers[-2](encoded_input)
decoder_layer = autoencoder.layers[-1](h2_layer)
decoder = keras.Model(encoded_input, decoder_layer)


# compile network
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# normalise and flatten data
train_data = (train_data/255).reshape((-1, 28*28))
test_data = (test_data/255).reshape((-1, 28*28))


# training
autoencoder.fit(train_data, train_data,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(test_data, test_data))


# make predictions
encoded_imgs = encoder.predict(test_data)
decoded_imgs = decoder.predict(encoded_imgs)

n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(test_data[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.tight_layout()
plt.savefig('ae_predictions.png', dpi=300)


# show latent space
x_arr = np.arange(8)
```

```python
y_arr = np.arange(2, 4)
mesh = np.array(np.meshgrid(x_arr, y_arr))
combinations = mesh.T.reshape(-1, 2)

arrs = encoded_imgs.T
arrs = encoded_imgs.T
fig, axs = plt.subplots(7, 7, figsize=(15, 15))
label = False
for i in range(8):
    for j in range(7):
        ax = axs[i-1, j]
        if (j >= i):
            ax.axis('off')
            continue
        ax.axis('on')
        x = arrs[j]
        y = arrs[i]
        for n in range(10):
            k = np.where(labels == n)
            if not label:
                ax.scatter(x[k], y[k], label=n)
            else:
                ax.scatter(x[k], y[k])
        label = True
        if i == 7:
            ax.set_xlabel(rf"$x_{j}$")
        if j == 0:
            ax.set_ylabel(rf"$x_{i}$")
fig.legend(loc='center left', bbox_to_anchor=(0.3, 0.8))
plt.tight_layout()
plt.savefig('latent_space.png', dpi=300)


# Save models
autoencoder.save('autoencoder')
encoder.save('encoder')
decoder.save('decoder')


# part 2

# number of images = nx * ny
nx = 15
ny = 15
# how much of latent space we will plot
x_min = 0
x_max = 40
y_min = 0
y_max = 30

xs = np.linspace(x_min, x_max, nx)
ys = np.linspace(x_min, x_max, ny)

# will plot how these change
x0, x5 = np.meshgrid(xs, ys)
shape = x0.shape

# get constant values for each other node
x1 = np.full(shape, 20)
x2 = np.full(shape, 20)
x3 = np.full(shape, 10)
x4 = np.full(shape, 20)
x6 = np.full(shape, 10)
x7 = np.full(shape, 20)

full_grid = np.stack((x0, x1, x2, x3, x4, x5, x6, x7)).T

# reshape into big combined picture
full_grid = full_grid.reshape(nx*ny, 8)
decoded_grid = decoder.predict(full_grid)
decoded_grid = decoded_grid.reshape((nx, ny, 28, 28))
```

```python
decoded_grid = decoded_grid.swapaxes(1, 2)
decoded_grid = decoded_grid.reshape(28*nx, 28*ny)

fig, ax = plt.subplots(figsize=(10, 10))
extent = [x_min, x_max, y_min, y_max]
ax.imshow(decoded_grid, extent=extent)
ax.set_xlabel(r"$x_0$")
ax.set_ylabel(r"$x_5$")
plt.tight_layout()
plt.savefig('image_grid.png', dpi=300)
```