TensorFlow Autoencoder Tutorials Learnings:

* TensorFlow Workflow:
  + Preprocessing
    - Flattening / Shape anpassen
    - Normalisieren
    - Train/Test Split
    - Tf.data
      * Batches erstellen
      * Shufflen
  + Class erstellen und tf.keras.Model inheriten
    - \_\_init\_\_(self)
      * Super.\_\_init\_\_()
      * Netz definieren
    - Call(self,x)
      * Input durch Netz laufen lassen und Output returnen
  + Objekt der Klasse erstellen
  + Objekt.compile()
    - Wie und mit welchem Loss soll optimiert werden?
  + Objekt.fit()
    - Trainingsdaten, Anzahl Epochen, Shufflen ja/nein, Validierungsdaten,…
  + Objekt.summary()
    - Auch der einzelnen Subnetze! Zb. Objekt.encoder.summary()
    - Wie sehen Input und Output Shape aus?
    - Wie viele Params gibt es?
  + Output (Daten) inspizieren/visualisieren
* Init und Call:
  + Init initialisiere ich, wenn ich ein Objekt generiere (zB einen Simulator: main\_sim = MainSimulator(HierarchicalNormalSimulator())) und gebe in den Arguments an, wie das Objekt initialisiert werden soll
  + call ist dann das konkrete Ausführen dieses Objekts, in den Argumenten gebe ich die Details der Ausführung an (zB was soll wie verarbeitet werden)
    - <https://www.tensorflow.org/tutorials/text/nmt_with_attention?hl=en#write_the_encoder_and_decoder_model>: Bei Transformationen innerhalb von Call über jeder Transformation Kommentar mit: x shape after bla == (batch size, …,…)
* Super():
  + Über super() inherited meine neu kreierte Klasse Autoencoder alle Funktionen und Attribute von tf.keras.Model (wenn nur Tensforflow als tf importiert) oder wie hier von Model, wenn explizit importiert wurde (from tensorflow.keras.models import Model) 🡪 wir können dadurch compile(), evaluate(), etc nutzen!

class Autoencoder(Model):

def \_\_init\_\_(self, latent\_dim):

super(Autoencoder, self).\_\_init\_\_()

* Layers.Flatten():
  + Input wird in von multidimensional in eindimensional (langer Vektor) verwandelt
* Markdown Formatierung:
  + In TensorFlow Beispielen (v.a. Convolutional Variational Autoencoder) nachschauen, wie z.B. statistische Formeln etc dort formatiert werden
* Tf.GradientTape():
  + <https://www.youtube.com/watch?v=T8AW0fKP0Hs> Eager Execution
    - Tape records all the operations you executed, so that you can play it back when you computed the gradients
  + <https://www.youtube.com/watch?v=_u7AVsxANes> Custom Training Loops
    - Records everything you do in forward propagation, so you can then do backward propagation for the model weights
    - 2 Loops im Training Loop
      * For epoch in range(num\_epochs):
        + For batch\_idx, (x\_batch, y\_batch) in enumerate(ds\_train):

with tf.GradientTape() as tape:

1. Predictions auf Batch der Trainingsdaten (x\_batch) anwenden -> y\_pred

2. Loss durch Anwendung der Loss function auf y\_batch und y\_pred berechnen -> loss

Dann tape nutzen um Gradients zu berechnen (compute gradients oft he loss with respect to the trainable parameters/weights): tape.gradients(loss, model.trainable\_weights) (bei Stefan: model.trainable\_variables)

Anschließend Gradients auf trainable weights applyen -> Backpropagation step abgeschlossen

Abschließend state der model performance metric updaten, damit man weiß, wie sie in der epoch aussah

* + - * + dann außerhalb das Loops des Batches zum Abschluss der Epoch Performance Metricprinten 🡪 abschließend resetten, damit für nächste Epoch ready
  + <https://stackoverflow.com/questions/53953099/what-is-the-purpose-of-the-tensorflow-gradient-tape>
  + <https://www.tensorflow.org/tutorials/text/nmt_with_attention?hl=en#write_the_encoder_and_decoder_model>
    - Kann ich einfach wenn ich mehrere Netze trainiere, die Weights von allen in einer trainable variable kombinieren?  
      variables = encoder.trainable\_variables + decoder.trainable\_variables