Differential deep learning with autodiff and autoencoder

Joerg Zinnegger, jzinnegger@z-frm.com

February 2, 2021

Abstract

This paper builds on the differential machine learning concept and implementations developed by Brian Huge and Antoine Savine. It discusses alternative technical implementations that utilize the Keras model framework in Tensorflow 2. The implementation includes a bespoke Keras layers for an explicit backpropation and the utilization of the Tensorflow GradientTape. It addition an autoencoder is added to the network to improve the calibration in absence of a differential PCA pre-processing.

1 Illustration of pathwise differential learning

The differential machine learning approach by Brian Huge and Antoine Savine [1], [2] is a generic deep neural network (DNN) approach to approximate pricing functions (see also Github *differential-machine-learning* [3]). The computational efficiency results from a combination of automatic adjoint differentation in the generation of the training data and the mirroring training of a differential deep network.

Essentially the DNN represents a regression over the Monte Carlo paths of the pricing engine. The use of bespoke regressions over simulated paths is a well known approach in the context of American Monte Carlo methods. DNNs have more general capabilities to approximate multivariate functions and the applications are not confined to a pathwise approach only. A discussion of the theoretical foundations, including the *Universal Pricing Theorem*, can – for example – be found in Horvath, at al. [4]. Huge and Savine [2] motivate that the pathwise training using a *mean squared error* (MSE) loss function (approximately) converges to the true value.

The novel feature in differential learning is the simultaneous calibration of the network on pathwise fair values (discounted cash flows) and pathwise differentials. The simultaneous calibration requires that the derivatives are obtained from the network during each cycle of the learning loop, in other words, that the network outputs include fair value as well as derivatives predictions. In the differential learning approach the differentials are obtained by algorithmic differentation with respect to the input parameters. The algorithmic differentation in the original work is implemented as a network extension, coined the *twin net*.

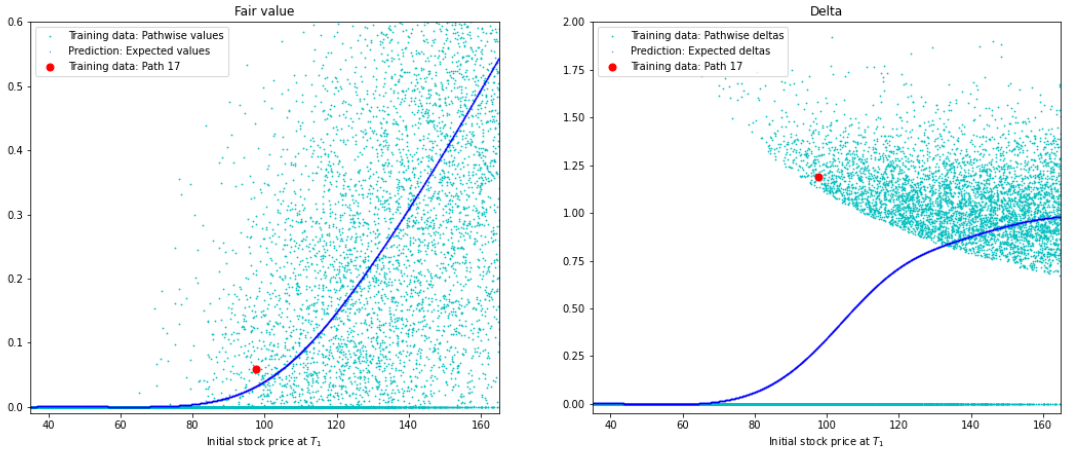

Figure 1: Regression on pathwise values and deltas

Figure 1 illustrates the regression approach in the classical Black Scholes set-up. The cyan dots represent the paths in the training set. Each path holds a realisation of the pathwise fair values (chart on the left) and corresponding pathwise delta (chart on the right). In the specific example fair values and deltas are weighted equally in the combined loss function. The blue lines are the predictions from the differential network.

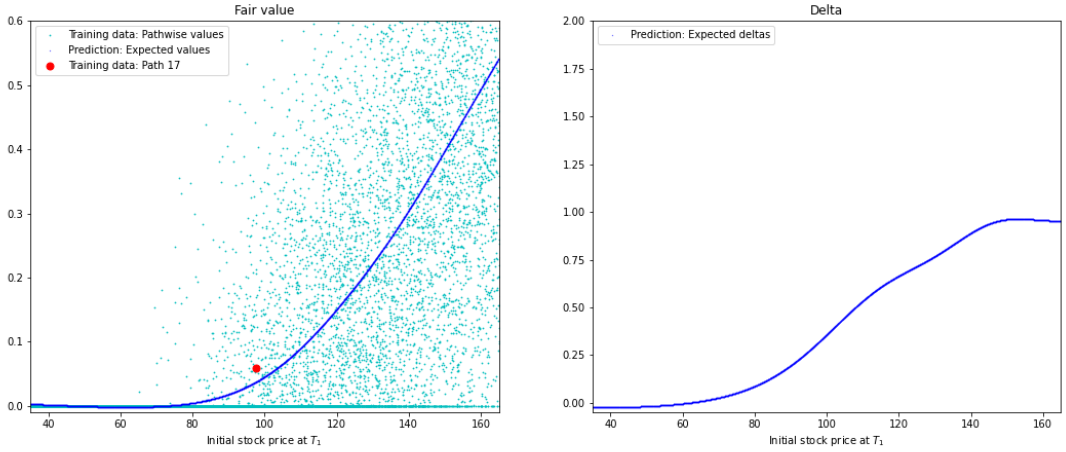

Figure 2: Regression on pathwise values without calibration on derived delta

In figure 2 the network is calibrated on the fair values only. The network predicts fair values and derivatives based on the functional relationship between the two outputs irrespective of the chosen calibration. However the quality of the derivative prediction is lower without the additional differential data. On the other extreme the network could be calibrated on differentials only.

Following the properties of differential equations the fair values can only predicted up to an unknown constant offset.

A note on the simulated paths: the original work and subsequent results use antithetic variables for the path generation. For illustration purposes no antithetic variables have been used in the Black Scholes example.

The code discussed in this paper is available as a Colab notebook on GitHub: https://github.com/jzinnegger/differential-ml.

2 Alternative technical implementation of the backpropagation

2.1 Twin net and backpropagation layer

The efficient calculation of differentials in the twin net is based on an (explicit) algorithmic differentation; or (explicit) backpropagation in machine learning terms. The differentation is performed in reverse order for each layer of the feedforward network.

This paragraph describes a possible implementation of the twin net utilizing the functionality of *Tensorflow 2*[1] and specifically the *Keras* model abstraction in Tensorflow.

In Keras the feedforward network can be modelled as sequence of *Dense* layers that implement the feedforward equation. Taking the notation from [2] the equations of the standard feedforward network are:

$$\begin{aligned} z_0 &= x \\ z_l &= g_{l-1}(z_{l-1})w_l + b_l \quad , l = 1, \ldots, L \\ y &= z_L \end{aligned}$$

where w_l, b_l are the weights and biases, g_l is the activation function for the layer $l = 1, \ldots, L$ and x, y are the model inputs and outputs. The notation implies that the activation is performed on the node input of the subsequent layer whilst the standard convention (also implemented in Keras) performs the activation on the node output. The modification is required for the backpropagation described in the next step.

Again following [2] the equations for the explicit backpropagation with respect to the model inputs can be derived:

$$\begin{aligned} \bar{z}_L &= \bar{y} = 1 \\ \bar{z}_{l-1} &= \left(\bar{z}_l w_l^T\right) \circ g'_{l-1}\left(z_{l-1}\right) \quad , l = L, \ldots, 1 \\ \bar{x} &= \bar{z}_0 \end{aligned}$$

with $\bar{z}_l = \delta y / \delta z_l$ and $\circ$ the elementwise product. As a direct result from the application of the chain rule, the non-activated output z_l of the feedforward are required as an input to the backpropagation equations.

The activation $g(.)$ needs to be differentiated once for the backpropagation, leading to $g'(.)$. In the example the default *relu* activation function is replaced by the differentiable *softplus* function and their differential *sigmoid*.

[1] see https://www.tensorflow.org

The backpropagation equations are not part of a standard Keras layer and are implemented as a bespoke layer. As the weights are shared with the feedforward network, the backpropagation layer consumes the corresponding instance of the feedforward network (a standard *Dense* layer) at instanciation.

```
class BackpropDense(tf.keras.layers.Layer):
    def __init__(self, reference_layer, **kwargs):
      super(BackpropDense, self).__init__(**kwargs)
      self.units = None
      self.ref_layer = reference_layer # weights of ref layer 'collected' by tensorflow

    def call(self, gradient, z):
        if z is not None:
            # essential backprop equation
            gradient = tf.matmul(gradient, tf.transpose(self.weights[0])) * tf.math.sigmoid(z)
        else:
            gradient = tf.matmul(gradient, tf.transpose(self.weights[0]))
        return gradient
```

We are now ready to translate the forward and backward equations into Keras layers. Note that the default activation of the dense layer is suppressed and initialisation takes place at the input of the subsequent layer.

```
// instance of a layer of dense feedforward network
...
layer_3 = layers.Dense(20, kernel_initializer='glorot_normal', activation = None, name='FWD_L3')
...

// forward network
...
x3 = layer_3(layers.Activation('softplus', name="Act_2")(x2))
...

// backward network
...
zbar = BackpropDense(layer_3,name='Bck_L3')(zbar, x2)
...
```

2.2 Autodiff and autoencoder

Backpropagation is a fundamental component in any deep learning environment as the weight updates are performed via a stochastic gradient approach. In tensorflow the backpropagation is labelled *reverse automatic differentation*. It comes with a tape for automatic algorithmic differentation, the *tf.GradientTape*, that implements automatic differentation for the standard tensorflow operators.

The GradientTape can be used to add a generic automatic differentation with respect to the input parameters of the model. The tape is a second inner tape to the standard backpropagation used in

the training loop. The inner gradient tape records the input variables (in contrast to the weights in the outer tape) with respect to the model output, i.e. the fair value (in contrast to the value of the loss function in the outer tape).

The inner tape can be embedded in a bespoke autodiff layer that consumes the whole feedforward network at instanciation. The weights of the feedforward network are automatically 'collected,' i.e. the autodiff layer shares the weights with the feedforward network.

```
class AutodiffLayer(tf.keras.layers.Layer):
    def __init__(self, fwd_model, **kwargs):
      super(AutodiffLayer, self).__init__(**kwargs)
      self.units = None
      self.fwd_model = fwd_model # weights of ref layer 'collected' by tensorflow

    def call(self, input):
        with tf.GradientTape(watch_accessed_variables=False) as tape:
            tape.watch(input)
            pred_value = self.fwd_model(input)

        # Get the gradients of the loss w.r.t to the pricing inputs
        gradient = tape.gradient(pred_value, input)

        return gradient
```

The generic reverse autodiff can be added to a keras model by one or two lines of code. The forward model and the autodiff model are stacked together via a functional model, sharing the same input.

```
...
fwd_model = tf.keras.models.Model(inputs=input_1, outputs=y_pred)

autodiff_layer = AutodiffLayer(fwd_model, name='dydx_pred')
dydx_pred = autodiff_layer(input_1)

model = tf.keras.models.Model(inputs=input_1, outputs=[y_pred, dydx_pred], name='Autodiff_Autoencoder')
```

Running the keras model summary for a stylized input with 3 parameters, we see that the forward model and the backward model (AutodiffLayer) each have 1,770 parameters. As the weights of the backward model are shared with the forward model the total number trainable paramters is still 1,770.

```
Model: "Autodiff_Autoencoder"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_3 (InputLayer)         [(None, 3)]               0
_________________________________________________________________________________________________
auto_encoder (Autoencoder)   (None, 1)                 449         input_3[0][0]
_________________________________________________________________________________________________
FWD_L1 (Dense)               (None, 20)                40          auto_encoder[0][0]
_________________________________________________________________________________________________
```

```
FWD_L2 (Dense)                  (None, 20)           420         FWD_L1[0][0]
__________________________________________________________________________________________________
FWD_L3 (Dense)                  (None, 20)           420         FWD_L2[0][0]
__________________________________________________________________________________________________
FWD_L4 (Dense)                  (None, 20)           420         FWD_L3[0][0]
__________________________________________________________________________________________________
y_pred (Dense)                  (None, 1)            21          FWD_L4[0][0]
__________________________________________________________________________________________________
dydx_pred (AutodiffLayer)       (None, 3)            1770        input_3[0][0]
==================================================================================================
Total params: 1,770
Trainable params: 1,770
Non-trainable params: 0
```

The forward model has been extended by an additional first layer that represents a bespoke autoencoder with eight latent nodes. The motivation is to reduce the dimensionality on an ad-hoc basis and is further discussed in chapter 4. The generic autodiff layer consumes the forward model including the autoencoder without special considerations.

3 Basket option

3.1 Prediction of fair values and deltas

The original implemementation by [2] includes an example of an european option on an equity basket priced with a Bachielier model. The original code is re-used to generate training examples for a basket with 20 constituents. The basket option is used for a technical demonstration of a multivariate input. However it is not intrinsically multivariate and an analytical solution is available. Therefore the example is not necessarily representative for real portfolios.

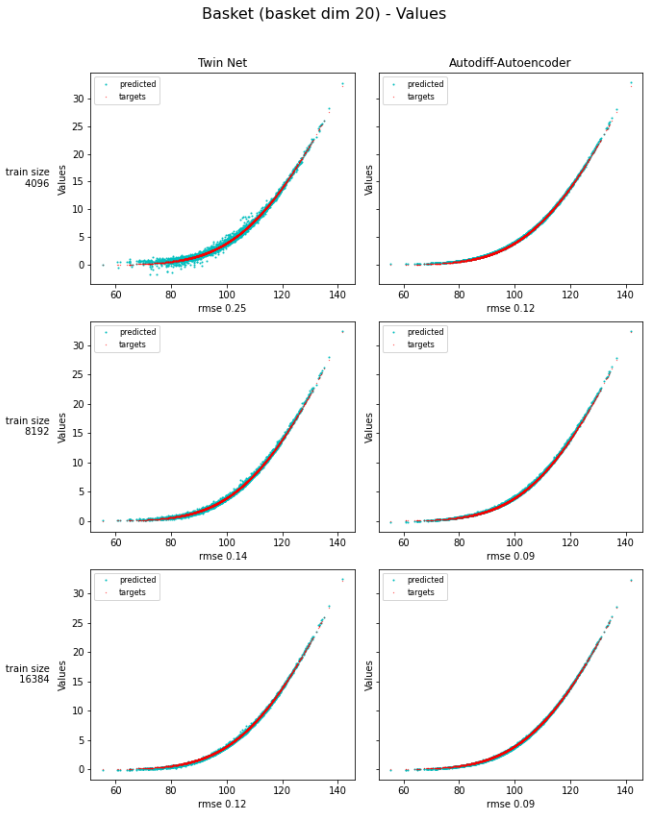

Figure 3: Predictions of fair values with different training sizes

Figure 3 shows a similiar performance of the *twin net* and the *autodiff-autoencoder* net for the fair value predictions with different training sizes. The cyan dots are the predicted fair values for the corresponding basket values. The red line is true value derived by the analytic formula.

In the training the number of sixteen steps per epoch is kept constant, i.e. an increase in the training sample translates into a proportional increase of the batch size. Keeping the batch size constant is potentially preferable for very large training samples.

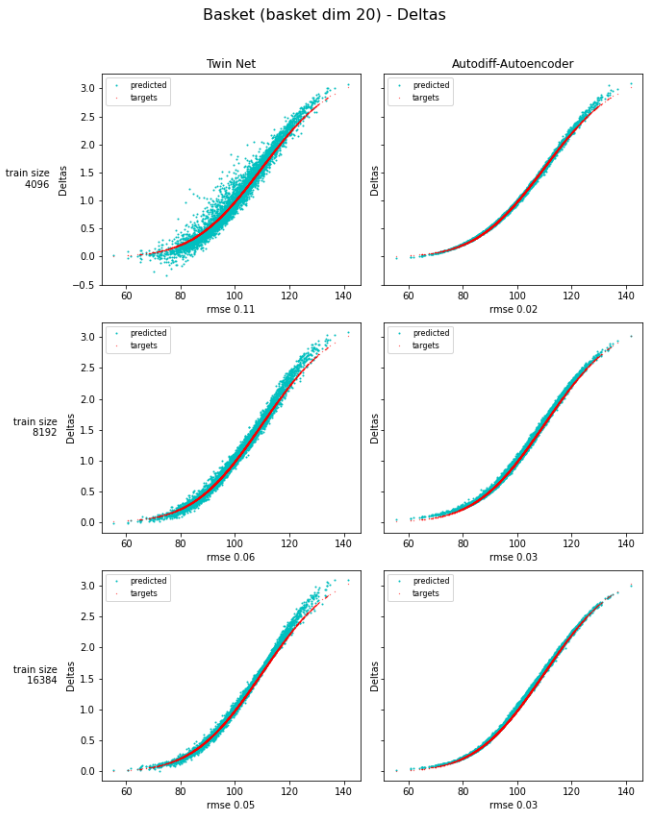

Figure 4: Predictions of deltas with different training sizes

3.2 Learning rate schedule and validation loss

The original approach makes use of a bespoke warm-up schedule that increases the learning in the first epochs and decreases thereafter. This type schedule is used for the training of the twin-net. Alternatively an of the shelve *inverse time decay* schedule is used for the training of the autodiff-autoencoder. The initial learning rate for the inverse time decay has been determined by manually increasing a constant learning rate. The chosen initial learning rate is almost a magnitude lower as the peak rate in the warm up schedule. This might imply a slower convergence for the autodiff-autoencoder.

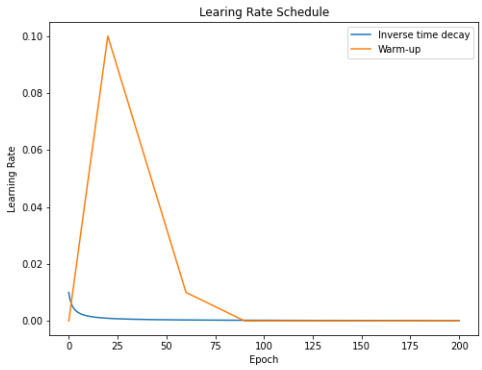

Figure 5: Learning rate schedule for twin-net and autodiff net

Figure 5 exhibits the time decay for the learning rate schedules. The convergence is typically assessed by comparing the evolution of the training loss with the validation loss. In the standard approach the validation set is a subset of the data that is not included in the training data. In the case of pathwise learning the training data will always be noisy by construction and a single path will in general not represent the true value of the derivative.

For the illustration of the convergence it is convenient to redefine the validation loss as the difference between the prediction and the true fair value. Both values are independent of the path and conditional on the input parameters only. True targets are in general not available for training.

Figure 6 shows the validation loss of the twin net (blue colour) versus the autodiff-autoencoder (red colour). The figure is a snapshot from the interactive tensorboard platform to which the log files of the experiments have been uploaded[2].

For the selected example the warm-up schedule does not offer an improved convergence. However, be reminded that the example does not necessarily generalize to real world portfolios.

[2]https://tensorboard.dev/experiment/V9w8TLm1RjyVeaDdwIBAaA/

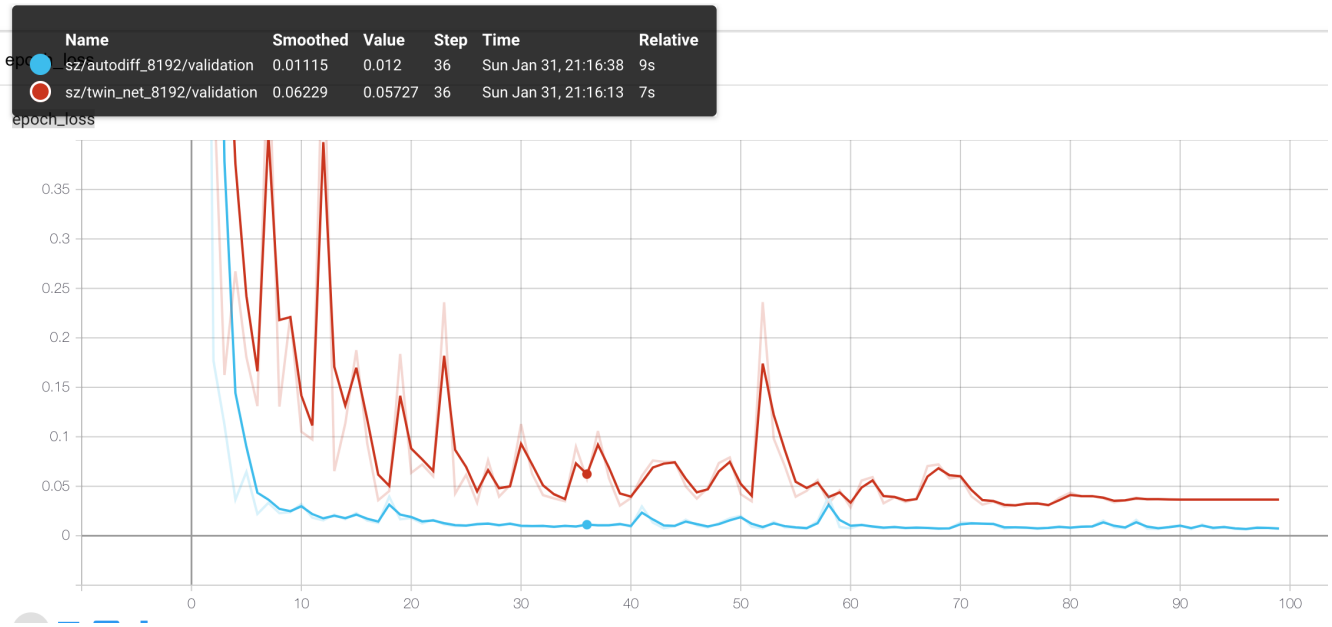

Figure 6: Tensorboard snapshot: Convergence of prediction loss for twin-net and autodiff-autoencoder

4 Higher dimensions and autoencoder

The original paper by Huge and Savine [2] adresses potential numerical instabilities in the calibration of the differential network and elaborates two mitigating techniques. The first mitigant is the extension of the network by a classical regression that serves as a theoretical backstop in case the network does not calibrate properly. The second mitigant adresses the higher dimensions explicitely. The concept is to reduce the input to the (latent) risk factors that drive the valuation, thereby reducing the model to it's intrinsic dimensionality. This is achieved by a principal component analysis performed on the values and differentials, called a *differential PCA*. The differential PCA is intended as a pre-processing step on the inputs.

Instead of the PCA we add an autoencoder as the first layer. It is well known that autoencoders can span the same subspace as a PCA, but do not have identical loading vectors. The recovery of the loading vectors from the autoencoder weights is adressed in [5]. Following the discussion on *CrossValidated* (see [6], [7]) there is no computational benefit in implementing a PCA via an autoencoder.

Adding an autoencoder as the first layer in the feedforward network can be seen as an ad-hoc attempt to extend the network capabilities and calibration stability without implementing the differential PCA pre-processing step. As the generic autodiff layer consumes the forward model including the autoencoder, the differential approach extends to the autoencoder without additional considerations.

The implementation of the autoencoder is straightforward. In contrast to the differential PCA

no further statistical insights are available to determine the appropriate latent dimension of the hidden layer.

```
class Autoencoder(tf.keras.layers.Layer):
    def __init__(self, input_dim, latent_dim, **kwargs):
        super(Autoencoder, self).__init__(**kwargs)
        self.latent_dim = latent_dim
        self.encoder = tf.keras.Sequential([
            layers.Input(shape=(input_dim,)),
            layers.Dense(20,  kernel_initializer='glorot_normal', activation = 'linear'),
            layers.Dense(latent_dim,  kernel_initializer='glorot_normal', activation = 'softplus'),
        ])
        self.decoder = tf.keras.Sequential([
            layers.Dense(20,  kernel_initializer='glorot_normal', activation = 'linear'),
            layers.Dense(1,  activation = 'softplus'),
        ])

    def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

Figure 7 illustrates the impact of the autoencoder on the calibration quality at higher dimensions. The number of basket constituents increases from 10 to 100 names. The graph shows the delta with respect to one underlying stock, whereby the x-axis is the corresponding level of the basket. This is a slice in one delta dimension, the actual calibration is performed on the weighted sum of the MSE loss function over all differentials. For the twin-net no (differential) PCA pre-processing has been performed. In case of no additional pre-processing the ad-hoc autoencoder layer adds to the stability of the calibration.

Another way is to look at the validation loss over all differentials, i.e. the aggregated MSE of all predicted differentials versus true differentials. The aggregated MSE is a weighted sum of the MSE losses for values and differentials.

In figure 8 the validation loss for the autodiff-autoencoder (blue) converges whereas the twin-net (read) diverges. The specific choice of the batch size in combination with the different learning rate schedule might contribute to the calibration instability, but the general issue is also reported in [2].

Even without elaborating on the theoretical foundations (see again [2]) it is important to keep in mind that the setup of differential learning is quite restrictive with respect to the typical approaches to improve the calibration. A core assumption in (differential) learning is the MSE loss function. Other loss functions or L1/L2 regularizations that add to the MSE loss function can introduce a non vanishing bias and a systematic pricing error.

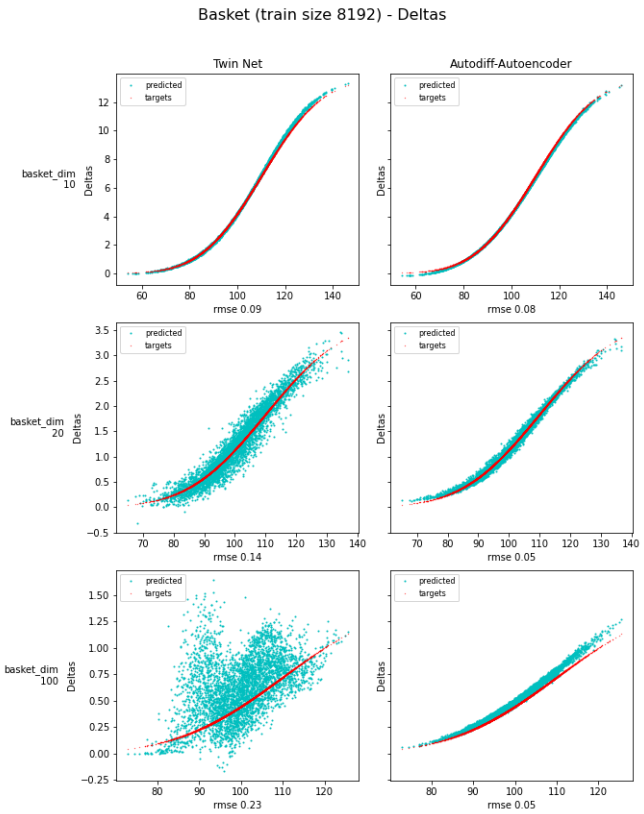

Figure 7: Impact of autoencoder at higher dimensions

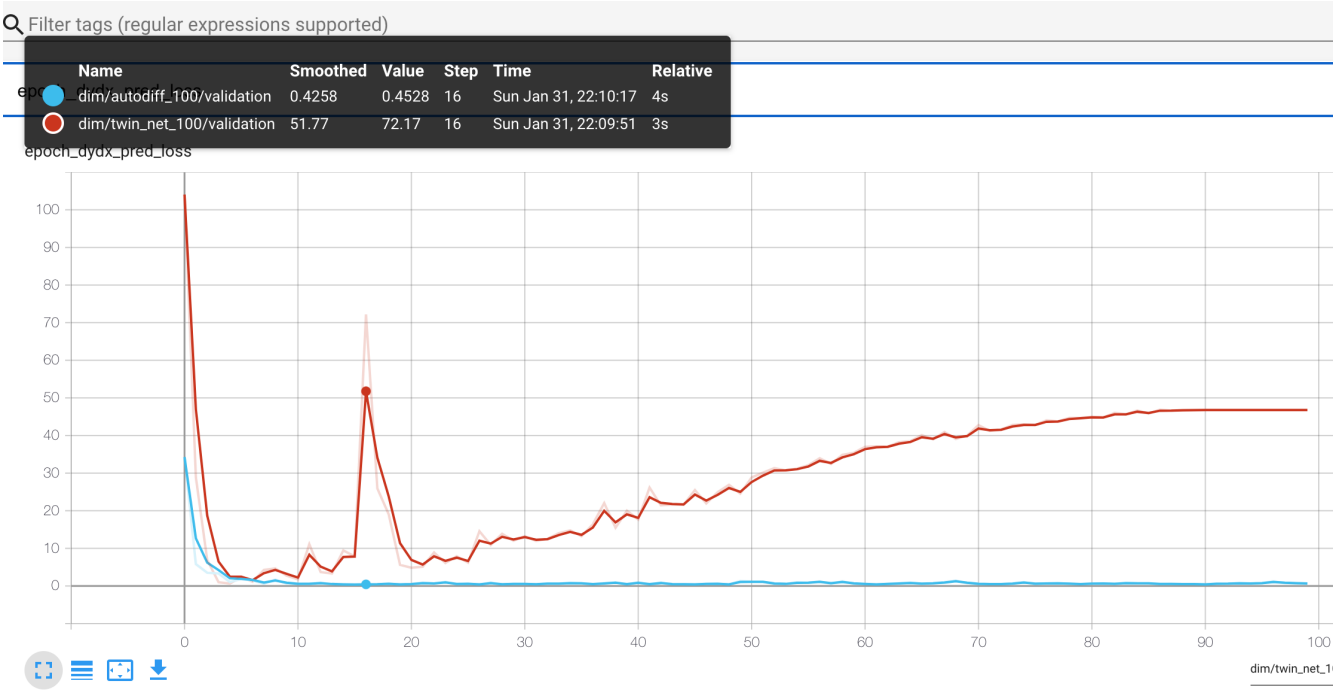

Figure 8: Tensorboard snapshot: Aggregated validation loss for a basket with 100 names (MSE of predicted vs true deltas) with autodiff-autoencoder (blue) and twin net (red)

5 Appendix: Twin net

The full structure of the twin net:

```
# feedforward

 # init fwd layers explicitely, convenient for reference in backprop
layer_1 = layers.Dense(20, kernel_initializer='glorot_normal', activation = None, name='FWD_L1')
layer_2 = layers.Dense(20, kernel_initializer='glorot_normal', activation = None, name='FWD_L2')
layer_3 = layers.Dense(20, kernel_initializer='glorot_normal', activation = None, name='FWD_L3')
layer_4 = layers.Dense(20, kernel_initializer='glorot_normal', activation = None, name='FWD_L4')
layer_5 = layers.Dense(1, kernel_initializer='glorot_normal', activation = 'linear', name='y_pred')


input_1 = layers.Input(shape=(input_dim,))
x1 = layer_1(input_1)
x2 = layer_2(layers.Activation('softplus', name="Act_1")(x1))
x3 = layer_3(layers.Activation('softplus', name="Act_2")(x2))
x4 = layer_4(layers.Activation('softplus', name="Act_3")(x3))
y_pred = layer_5(layers.Activation('softplus', name="Act_4")(x4))

 # backprop

 # backprop has no trainable weights, only references to layers in forward net
```

```
grad = BackpropDense(layer_5,name='Bck_L1')(tf.ones_like(y_pred), x4)
grad = BackpropDense(layer_4,name='Bck_L2')(grad, x3)
grad = BackpropDense(layer_3,name='Bck_L3')(grad, x2)
grad = BackpropDense(layer_2,name='Bck_L4')(grad, x1)
dydx_pred = BackpropDense(layer_1,name='dydx_pred')(grad, None)

model = tf.keras.models.Model(inputs=input_1, outputs=[y_pred, dydx_pred], name='Twin_Net')
```

and the respectiv model summary generated by keras:

```
Model: "Twin_Net"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 3)]          0
__________________________________________________________________________________________________
FWD_L1 (Dense)                  (None, 20)           80          input_1[0][0]
__________________________________________________________________________________________________
Act_1 (Activation)              (None, 20)           0           FWD_L1[0][0]
__________________________________________________________________________________________________
FWD_L2 (Dense)                  (None, 20)           420         Act_1[0][0]
__________________________________________________________________________________________________
Act_2 (Activation)              (None, 20)           0           FWD_L2[0][0]
__________________________________________________________________________________________________
FWD_L3 (Dense)                  (None, 20)           420         Act_2[0][0]
__________________________________________________________________________________________________
Act_3 (Activation)              (None, 20)           0           FWD_L3[0][0]
__________________________________________________________________________________________________
FWD_L4 (Dense)                  (None, 20)           420         Act_3[0][0]
__________________________________________________________________________________________________
Act_4 (Activation)              (None, 20)           0           FWD_L4[0][0]
__________________________________________________________________________________________________
y_pred (Dense)                  (None, 1)            21          Act_4[0][0]
__________________________________________________________________________________________________
tf.ones_like (TFOpLambda)       (None, 1)            0           y_pred[0][0]
__________________________________________________________________________________________________
Bck_L1 (BackpropDense)          (None, 20)           21          tf.ones_like[0][0]
                                                                 FWD_L4[0][0]
__________________________________________________________________________________________________
Bck_L2 (BackpropDense)          (None, 20)           420         Bck_L1[0][0]
                                                                 FWD_L3[0][0]
__________________________________________________________________________________________________
Bck_L3 (BackpropDense)          (None, 20)           420         Bck_L2[0][0]
                                                                 FWD_L2[0][0]
__________________________________________________________________________________________________
Bck_L4 (BackpropDense)          (None, 20)           420         Bck_L3[0][0]
                                                                 FWD_L1[0][0]
__________________________________________________________________________________________________
dydx_pred (BackpropDense)       (None, 3)            80          Bck_L4[0][0]
==================================================================================================
Total params: 1,361
```

```
Trainable params: 1,361
Non-trainable params: 0
----------------------------------------------------------------------------------------------------
```

Bibliography

[1] B. Huge and A. Savine, "Differential machine learning: The shape of things to come," *Risk Magazin*, Oct. 2020 [Online]. Available: https://www.risk.net/cutting-edge/banking/7688441/differential-machine-learning-the-shape-of-things-to-come

[2] B. Huge and A. Savine, "Differential Machine Learning," *arXiv:2005.02347 [cs, q-fin]*, Sep. 2020 [Online]. Available: http://arxiv.org/abs/2005.02347. [Accessed: 24-Jan-2021]

[3] B. Huge and A. Savine, "Github: Differential-machine-learning," *GitHub*. [Online]. Available: https://github.com/differential-machine-learning. [Accessed: 02-Feb-2021]

[4] B. Horvath, A. Muguruza, and M. Tomas, "Deep Learning Volatility," *arXiv:1901.09647 [q-fin]*, Aug. 2019 [Online]. Available: http://arxiv.org/abs/1901.09647. [Accessed: 29-Jan-2021]

[5] E. Plaut, "From Principal Subspaces to Principal Components with Linear Autoencoders," *arXiv:1804.10253 [cs, stat]*, Dec. 2018 [Online]. Available: http://arxiv.org/abs/1804.10253. [Accessed: 01-Feb-2021]

[6] D. (https://stats.stackexchange.com/users/16922/daemonmaker), "What are the differences between PCA and autoencoder?" [Online]. Available: https://stats.stackexchange.com/q/120295

[7] D. DeltaIV (https://stats.stackexchange.com/users/58675/deltaiv), "What are the differences between PCA and autoencoder?" [Online]. Available: https://stats.stackexchange.com/q/357247