Deep Learing TenforFlow code explained.

The solution of a canonical consumption-saving problem in the Euler residual minimization approach with borrowing constraints and four exogenous stochastic shocks.

```
import TenforFlow as tf
import numpy as np
from math import sqrt
from matplotlib import pyplot as plt
from tqdm import tqdm as tqdm
# tqdm is a nice library to visualize ongoing loops
import datetime
# followint lines are used for indicative typing
from typing import Tuple
class Vector: pass
```

Parameterization of the agency's decision function with a multilayer neural network, then performing training using stochastic optimization. In each iteration, training the model on one or few grid points randomly drawn from the state space (instead of conventional fixed grid with a large number of grid points by a discretization of the state-space)

The objective function-the sum of squared residuals in the Euler equation- has two expectation operators, one with respect to current state variables, the other with repect to future state variables.

Merges two expectation operators into one. By doing this, eliminates the correlation between terms and also reduces the cost of training deep neural networks.

Consumption-saving problems:

$$\max_{\{c_t, w_{t+1}\}_{t=0}^{\infty}} E_0\left[\sum_{t=0}^{\infty} \exp(\delta_t)\beta^t u\left(c_t\right)\right]$$

$$\text{s.t. } w_{t+1} = \left(w_t - c_t\right)r\exp(r_{t+1}) + \exp(y_{t+1}),$$

$$c_t \leq w_t,$$

c_t is consumption; w_t is the beginning-of-period cash-on-hand; $\beta \in [0,1)$ is a subjective discount factor; $r \in (0,1/\beta)$ is a constant interest rate; and initial condition (z, w) is given. Binding condition is consumption cannot exceed cash-on-hand. Four different exogenous state variables, shocks to the interest rate r_t, discount factor δ_t, transitory component of income q_t, and permanent component of income p_t. Total income is $y_t = \exp(p_t)\exp(q_t)$. All exogenous variables follow AR(1) processes

$$y_{t+1} = \rho_y y_t + \sigma_y \epsilon^y_{t+1},$$
$$p_{t+1} = \rho_p p_t + \sigma_p \epsilon^p_{t+1},$$
$$r_{t+1} = \rho_r r_t + \sigma_r \epsilon^r_{t+1},$$
$$\delta_{t+1} = \rho_\delta \delta_t + \sigma_\delta \epsilon^\delta_{t+1},$$

where error term follows N(0,1), utility function follows Cobb-Douglas utility function

$$u(c_t) = \frac{1}{1-\gamma}\left(c_t^{1-\gamma} - 1\right)$$

```
β = 0.9
γ = 2.0
# σ = 0.1
# ρ = 0.9
σ_r = 0.001
ρ_r = 0.2
σ_p = 0.0001
ρ_p = 0.999
σ_q = 0.001
ρ_q = 0.9
σ_δ = 0.001
ρ_δ = 0.2
rbar = 1.04
```

y, p, r, δ are drawn from ergodic distribution (AR(1) process z with autocorrelation ρ and conditional stdev σ, the ergodic distribution is normal with zero mean and stdev $\sigma_z = \sigma/\sqrt{1-\rho^2}$. w is drawn from a uniform distribution within min/max of w).

```
σ_e_r = σ_r/(1-ρ_r**2)**0.5
σ_e_p = σ_p/(1-ρ_p**2)**0.5
σ_e_q = σ_q/(1-ρ_q**2)**0.5
σ_e_δ = σ_δ/(1-ρ_δ**2)**0.5
wmin = 0.1
wmax = 4.0
```

In recursive form, the solution is Kuhn-Tucker(KT) conditions

$$a \geq 0, \quad b \geq 0 \quad and \quad ab = 0,$$

a is the share of wealth that goes into savings and b is the Lagrange multiplier

$$a \equiv w - c,$$

$$b \equiv u'(c) - \beta r \bar{E}_{\epsilon}\left[u'\left(c'\right)\exp\left(\delta' - \delta + r'\right)\right].$$

If $b = 0$, then the KT conditions lead to the Euler equation. As inequality constrains are not directly compatible with the deep learning framework developed in the paper, reformulating the KT conditions as a set of equations that hold with equality. Using a smooth representation of the KT conditions, called the Fischer-Burmeister (FB) function, which is differentiable and equivalent to KT conditions when equals to 0.

$$\Psi_{FB}(a, b) = a + b - \sqrt{a^2 + b^2} = 0$$

For numerical treatment, rewrite FB into

$$\Psi_{FB}(1 - \zeta, 1 - h) = (1 - \zeta) + (1 - h) - \sqrt{(1 - \zeta)^2 + (1 - h)^2} = 0$$

ζ, h are respectively the consumption share and normalized Lagrange multiplier $\zeta \equiv \frac{c}{w}$

$$h \equiv \bar{\beta r E}_{\epsilon}\left[\frac{u'\left(c'\right)}{u'(c)}\exp\left(\delta' - \delta + r'\right)\right]$$

$\zeta \sim [0,1]$, which is convenient for defining neural network. h is always positive and is normalized to be around one.

```
min_FB = lambda a,b: a+b-tf.sqrt(a**2+b**2)
```

$$\begin{pmatrix}\zeta \\ h\end{pmatrix} = \varphi(s; \theta)$$

where $s = (r, \delta, q, p, w)$ is a 5 dimensional state space. A traditional econometric method is to approximate an unknown function φ using some flexible function family $\varphi(\ldots; \theta)$ parameterized by a vector of coefficients θ, a polynomial family. Neural networks are just a special family of approximating functions, which have a nonlinear dependence of the approximation function on the coefficients θ. Using TenforFlow's keras, multilayer perceptron: a 2 hidden layers 32x32x32x2 network with relu activation functions and linear outputs.

```
layers = [
    tf.keras.layers.Dense(32, activation='relu', input_dim=5,
bias_initializer='he_uniform'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(2, activation=tf.keras.activations.linear)
```

```
]
perceptron = tf.keras.Sequential(layers)
tf.keras.utils.plot_model(perceptron, to_file='model.png', show_shapes=True)
```

Then, create the decision rule which takes as input 5 vectors of the same size n for the states r, δ, q, p, w and returns two vectors of size n, ζ, h respectively. Then use different nonlinear transformations for the two decision functions, sigmoid/logistic and exponential ones.

$$\varphi(s;\theta) = \begin{pmatrix} \frac{1}{1+e^{-\mathcal{N}_1(s;\theta)}} \\ \exp(\mathcal{N}_2(s;\theta)) \end{pmatrix}$$

where $\mathcal{N}_1, \mathcal{N}_2$ denote first and second component of neural network output, which ensures $\zeta \in [0,1), h > 0$.

```
def dr(r: Vector, δ: Vector, q: Vector, p: Vector, w: Vector)-> Tuple[Vector,
Vector]:

    # normalize exogenous state variables by their 2 standard deviations
    # so that they are typically between -1 and 1
    r = r/σ_e_r/2
    δ = δ/σ_e_δ/2
    q = q/σ_e_q/2
    p = p/σ_e_p/2

    # normalze income to be between -1 and 1
    w = (w-wmin)/(wmax-wmin)*2.0-1.0

    # prepare input to the perceptron
    s = tf.concat([_e[:,None] for _e in [r,δ,q,p,w]], axis=1) # equivalent to
np.column_stack

    x = perceptron(s) # n x 2 matrix

    # consumption share is always in [0,1]
    ζ = tf.sigmoid( x[:,0] )

    # expectation of marginal consumption is always positive
    h = tf.exp( x[:,1] )

    return (ζ, h)
```

As an illustration, plot the initial guess of decision rules against w. The coefficients of the perceptron are initialized with random values. TenforFlow in an eager mode (calculations are returned immediately)

```
wvec = np.linspace(wmin, wmax, 100)
# r,p,q,δ are zero-mean
ζvec, hvec = dr(wvec*0, wvec*0, wvec*0, wvec*0, wvec)
plt.plot(wvec, wvec, linestyle='--', color='black')
plt.plot(wvec, wvec*ζvec)
plt.xlabel("$w_t$")
plt.ylabel("$c_t$")
plt.title("Initial Guess")
plt.grid()
```

The unknown decision functions for ζ, h, two model's equations, the definition of normalized Lagrange multiplier and the FB function representing the KT conditions,

$$h = \beta r E_{\epsilon}\left[\frac{u'(c')}{u'(c)}\exp\left(\delta' - \delta + r'\right)\right]$$

$$\Psi_{FB}(1-\zeta, 1-h) = 0$$

where $\epsilon' = (\epsilon_r', \epsilon_\delta', \epsilon_q', \epsilon_p')$, then construct the residuals in the above two equations to minimize. For given vectors of current state $s = (r, \delta, q, p, w)$, next period shocks ϵ' to compute realized residuals:

$$R_1(s, \epsilon') = \beta r \frac{-u'(c')}{u'(c)}\exp\left(\delta' - \delta + r'\right) - h$$

$$R_2(s) = \Psi_{FB}(1-\zeta, 1-h),$$

$$w' = (w - c)r\exp(r) + \exp(y)$$

```
def Residuals(e_r: Vector, e_δ: Vector, e_q: Vector, e_p: Vector, r: Vector,
δ: Vector, q: Vector, p: Vector, w: Vector):

    # all inputs are expected to have the same size n
    n = tf.size(r)

    # arguments correspond to the values of the states today
    ζ, h = dr(r, δ, q, p, w)
    c = ζ*w

    # transitions of the exogenous processes
```

```
    rnext = r*ρ_r + e_r
    δnext = δ*ρ_δ + e_δ
    pnext = p*ρ_p + e_p
    qnext = q*ρ_q + e_q
    # (epsilon = (rnext, δnext, pnext, qnext))

    # transition of endogenous states (next denotes variables at t+1)
    wnext = tf.exp(pnext)*tf.exp(qnext) + (w-c)*rbar*tf.exp(rnext)

    ζnext, hnext = dr(rnext, δnext, qnext, pnext, wnext)
    cnext = ζnext*wnext


    R1 = β*tf.exp(δnext-δ)*(cnext/c)**(-γ)*rbar*tf.exp(rnext) - h
    R2 = min_FB(1-h,1-ζ)

    return (R1, R2)
```

Now, construct the objective function for minimization of the squared sum of two residuals in the two model's equations on a given 5-dimensional domain $s = (r, \delta, q, p, w)$,

$$\Xi(\theta) = E_s\left[\left(E_\epsilon\left[R_1(s,\epsilon')\right]\right)^2 + v\left(R_2(s)\right)^2\right]$$

v is the exogenous relative weights of the two residuals in the objective function. To be able to merge the two expectation operators into a single one, use a probability theory that two random variables a, b and are independent and follow the same distribution then$E[a]E[b] = E[ab]$ and first term of above equation can be separated into two equations.

$$E_{\epsilon_1}\left[R_1(s,\epsilon_1')\right]E_{\epsilon_2}\left[R_1(s,\epsilon_2')\right] = E_{\epsilon_1,\epsilon_2}\left[R_1(s,\epsilon_1')R_1(s,\epsilon_2')\right]$$

$$\Xi(\theta) = E_{s,\epsilon_1,\epsilon_2}\left[R_1\left(s,\epsilon_1'\right)R_1\left(s,\epsilon_2'\right) + v\left(R_2(s)\right)^{2\xi(\omega;\theta)}\right] \equiv E_\omega[\xi(\omega;\theta)]$$

where $\omega = (s, \epsilon_1', \epsilon_2')$. Writing the objective function of the deep learning method as a single expectation operator $E_\omega[\xi(\omega;\theta]$ of a function $[\xi(\omega;\theta]$ that depends on a vector-valued random variable ω. Then approximate

$$\Xi(\theta) \approx \Xi^n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\xi(\omega_i;\theta)$$

By draw n random draws of $\omega = (s, \epsilon_1^{'}, \epsilon_2^{'})$ and compute the average of the objective function

```
def Ξ(n): # objective function for DL training

    # randomly drawing current states
    r = tf.random.normal(shape=(n,), stddev=σ_e_r)
    δ = tf.random.normal(shape=(n,), stddev=σ_e_δ)
    p = tf.random.normal(shape=(n,), stddev=σ_e_p)
    q = tf.random.normal(shape=(n,), stddev=σ_e_q)
    w = tf.random.uniform(shape=(n,), minval=wmin, maxval=wmax)

    # randomly drawing 1st realization for shocks
    e1_r = tf.random.normal(shape=(n,), stddev=σ_r)
    e1_δ = tf.random.normal(shape=(n,), stddev=σ_δ)
    e1_p = tf.random.normal(shape=(n,), stddev=σ_p)
    e1_q = tf.random.normal(shape=(n,), stddev=σ_q)

    # randomly drawing 2nd realization for shocks
    e2_r = tf.random.normal(shape=(n,), stddev=σ_r)
    e2_δ = tf.random.normal(shape=(n,), stddev=σ_δ)
    e2_p = tf.random.normal(shape=(n,), stddev=σ_p)
    e2_q = tf.random.normal(shape=(n,), stddev=σ_q)

    # residuals for n random grid points under 2 realizations of shocks
    R1_e1, R2_e1 = Residuals(e1_r, e1_δ, e1_p, e1_q, r, δ, q, p, w)
    R1_e2, R2_e2 = Residuals(e2_r, e2_δ, e2_p, e2_q, r, δ, q, p, w)

    # construct all-in-one expectation operator
    R_squared = R1_e1*R1_e2 + R2_e1*R2_e2

    # compute average across n random draws
    return tf.reduce_mean(R_squared)


n = 128
v = Ξ(n)
v
```

```
v.numpy()
```

Finally, perform minimization of the objective function by solving the model using stochastic optimization – the stochastic gradient descent method, version called Adam.

```
θ = perceptron.trainable_variables
print( str(θ)[:1000] )
```

For the stochastic gradient descent, the updating rule would be

$$\theta \leftarrow \theta(1-\lambda) - \lambda \nabla_{\theta} \Xi_{n}(\theta)$$

where λ is a learning rate. For Adam, the learning rate evolves oer time and can be specific to each coefficient.

```
from TenforFlow.keras.optimizers import Adam, SGD
variables = perceptron.trainable_variables
optimizer = Adam()
# optimizer = SGD(λ=0.1) # SGD can be used in place of Adam
```

GradientTape functionality from TenforFlow to compute the gradient from the objective, and supply it to the optimizer.

```
@tf.function
def training_step():

    with tf.GradientTape() as tape:
        xx = Ξ(n)

    grads = tape.gradient(xx, θ)
    optimizer.apply_gradients(zip(grads,θ))

    return xx

def train_me(K):

    vals = []
    for k in tqdm(tf.range(K)):
        val = training_step()
        vals.append(val.numpy())

    return vals

# with writer.as_default():
results = train_me(50000)

# plot the empirical errors against the number of epochs
plt.plot(np.sqrt( results) )
plt.xscale('log')
plt.yscale('log')
plt.grid()
```

This graph represents the mean of the squared residuals. By taking the square root, 5 * 10^(-3) for L_2 norm will be obtained, which is quite accurate approximation for a model with 5 states variables.

The constructed decision rule follows

```
wvec = np.linspace(wmin, wmax, 100)
ζvec, hvec = dr(wvec*0, wvec*0, wvec*0, wvec*0, wvec)

plt.title("Multidimensional Consumption-Savings (decision rule)")
plt.plot(wvec, wvec, linestyle='--', color='black')
plt.plot(wvec, wvec*ζvec)
plt.xlabel("$w_t$")
plt.ylabel("$c_t$")
plt.grid()
```