In [165]:
```
import numpy as np
import tensorflow as tf

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Conv2D, Concatenate, Dropout, Subtract, \
                        Flatten, MaxPooling2D, Multiply, Lambda, Add, Dot
from tensorflow.keras.backend import constant
from tensorflow.keras import optimizers

from tensorflow.keras.layers import Layer
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input
from tensorflow.keras import initializers
from tensorflow.keras.constraints import max_norm
import tensorflow.keras.backend as K
import math

import matplotlib.pyplot as plt

import scipy.stats as scipy
from scipy.stats import norm
```

Problem (a)

under the risk neutral measure Q, where r = 0.01 and σ = 0.2. Compute the Black Scholes delta hedge as a function of time and stock price. (Consider the payoff of the straddle option as a sum of a call option and a put option. The delta hedge for the straddle option is the sum of the delta hedge for a call and a put.)

Construct functions for call and put delta

In [166]:
```
def delta_call(N,S0,strike,T,sigma,r, k):
    delta = scipy.norm.cdf((np.log(S0/strike)+(r+0.5*sigma**2)*(T-k*T/N))/(np.sqrt(T-k*T/N)*sigma))
    return delta

def delta_put(N,S0,strike,T,sigma,r, k):
    delta = scipy.norm.cdf((np.log(S0/strike)+(r+0.5*sigma**2)*(T-k*T/N))/(np.sqrt(T-k*T/N)*sigma))
    delta = delta - 1
    return delta
```

We choose S from 60 to 150 (extracting 100 later) to see what happen to out hedging portfolio

In [167]:
```
N=30 # time disrectization
S0=100 # initial value of the asset
strike=100 # strike for the call option
T=1.0 # maturity
sigma=0.2 # volatility in Black Scholes
r = 0.01 # interest rates
R=10 # number of Trajectories

s=np.arange(60,160,10) # range for the asset price to compute the hedging strategy
k=21 # choose k between 1 and N-1

truedelta_call = delta_call(N,s,strike,T,sigma,r, k)
truedelta_put = delta_put(N,s,strike,T,sigma,r, k)
true_delta = truedelta_call + truedelta_put
```

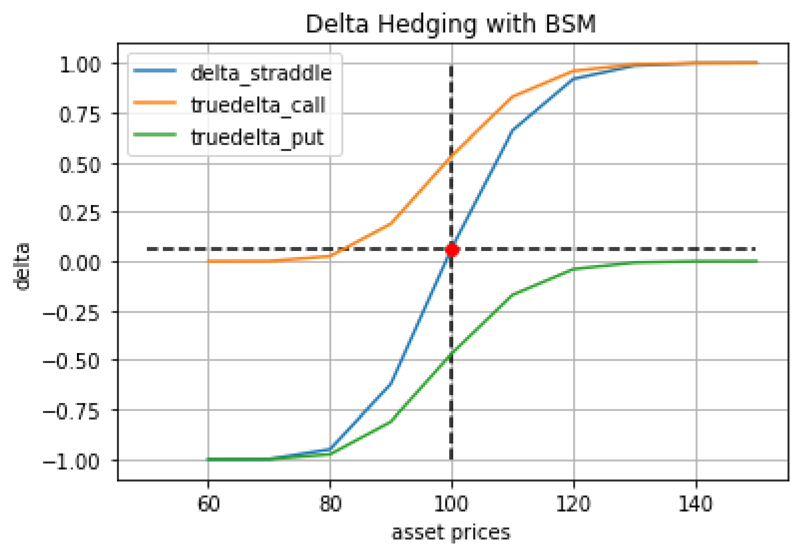
In [168]:
```
import pandas as pd
table = pd.DataFrame({"Asset Price" : s, "Call" : truedelta_call,"Put" : truedelta_put, "Straddle" : true_delta})
table
```

Out[168]:

	Asset Price	Call	Put	Straddle
0	60	0.000002	-0.999998	-0.999995
1	70	0.000752	-0.999248	-0.998496
2	80	0.025300	-0.974700	-0.949400
3	90	0.189525	-0.810475	-0.620950
4	100	0.532740	-0.467260	0.065479
5	110	0.829507	-0.170493	0.659013
6	120	0.959640	-0.040360	0.919279
7	130	0.993379	-0.006621	0.986758
8	140	0.999194	-0.000806	0.998388
9	150	0.999923	-0.000077	0.999845

We can see from the chart below that the straddle with S0 = K = 100 has the delta vert close to 0.

In [169]:
```
plt.plot(s,true_delta,s,truedelta_call,s,truedelta_put)
plt.xlabel('asset prices')
plt.ylabel("delta")
plt.title("Delta Hedging with BSM")
plt.grid(True)
plt.legend(["delta_straddle","truedelta_call","truedelta_put"])
plt.hlines(y = table.iloc[:,3][table.iloc[:,0] == 100], xmin = 50, xmax = 150, linestyles = "dashed")
plt.vlines(x = 100, ymin = -1, ymax = 1, linestyles = "dashed")
plt.plot(100, table.iloc[:,3][table.iloc[:,0] == 100])
plt.plot(100,table.iloc[:,3][table.iloc[:,0] == 100],'ro',label="point")
plt.show()
```

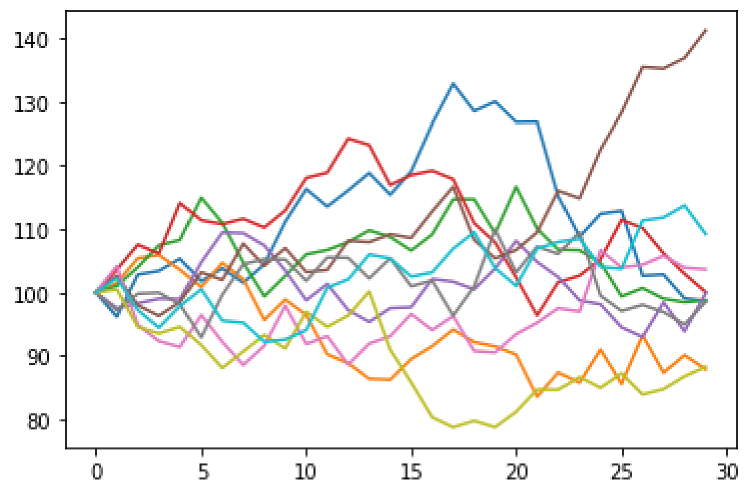


In [170]:
```
s=np.linspace(50,150,10) # range for the asset price to compute the hedging strategy
k=21 # choose k between 1 and N-1

truedelta_call = delta_call(N,s,strike,T,sigma,r, k)
truedelta_put = delta_put(N,s,strike,T,sigma,r, k)
true_delta = truedelta_call + truedelta_put
```

Problem(b)

Follow the “deep hedging BS keras.ipynb” to use deep neural networks to approximate the hedging strategy. Compare the deep hedging strategy to the Black Scholes hedging in (a).

In [171]:
```
logS= np.zeros((N,R))
logS[0,]=np.log(S0)*np.ones((1,R))

for i in range(R):
    for j in range(N-1):
        increment = np.random.normal((r - (sigma ** 2 / 2)) * (T / N),sigma*np.sqrt(T)/np.sqrt(N))
        logS[j+1,i] =logS[j,i]+increment

S=np.exp(logS)

for i in range(R):
   plt.plot(S[:,i])
plt.show()
```



In [172]:

```
import scipy.stats as stats
from scipy.stats import norm

#Blackscholes price

def BSM_put(S, K, sigma, t, r):
    d1 = (np.log(S / K) + (r + sigma ** 2 / 2) * t) / (sigma * t ** 0.5)
    d2 = (np.log(S / K) + (r - sigma ** 2 / 2) * t) / (sigma * t ** 0.5)
    Nd1 = stats.norm.cdf(-d1)
    Nd2 = stats.norm.cdf(-d2)
    put_price = Nd2 * K * np.exp(-r * t) - S * Nd1
    return put_price

def BSM_call(S, K, sigma, t, r):
    d1 = (np.log(S / K) + (r + sigma ** 2 / 2) * t) / (sigma * t ** 0.5)
    d2 = (np.log(S / K) + (r - sigma ** 2 / 2) * t) / (sigma * t ** 0.5)
    Nd1 = stats.norm.cdf(d1)
    Nd2 = stats.norm.cdf(d2)
    call_price = -Nd2 * K * np.exp(-r * t) + S * Nd1
    return call_price

priceBS1=BSM_call(S0,strike,sigma,T,r)
priceBS2=BSM_put(S0,strike,sigma,T,r)
priceBS = priceBS1 + priceBS2
print('Price of a Call option in the Black scholes model with initial price', S0, 'strike', strike, 'maturity', T , 'and volatility' ,
```

```
Price of a Call option in the Black scholes model with initial price 100 strike 100 maturity 1.0 and volatility 0.2 is equal to 15.
871620755136014
```

In [173]:

```
#Definition of neural networks for hedging strategies

m = 1 # dimension of price
d = 2 # number of layers in strategy
n = 32  # nodes in the first but last layers

# architecture is the same for all networks
layers = []
for j in range(N): # a neural network for each time step
    for i in range(d):
        if i < d-1:
            nodes = n
            layer = Dense(nodes, activation='tanh',trainable=True,
                      kernel_initializer=initializers.RandomNormal(0,1),#kernel_initializer='random_normal',
                      bias_initializer='random_normal',
                      name=str(i)+str(j))
        else:
            nodes = m
            layer = Dense(nodes, activation='linear', trainable=True,
                          kernel_initializer=initializers.RandomNormal(0,1),#kernel_initializer='random_normal',
                          bias_initializer='random_normal',
                          name=str(i)+str(j))
        layers = layers + [layer]
```

For the payoff, we should use (-r * T) to discount!

In [174]:
```
#Implementing the loss function
# Inputs is the training set below, containing the price S0,
#the initial hedging being 0, and the increments of the log price process
price = Input(shape=(m,))
hedge = Input(shape=(m,))

inputs = [price]+[hedge]


for j in range(N):
    strategy = price
    for k in range(d):
        strategy= layers[k+(j)*d](strategy) # hedging strategy at j , i.e. the neural network g_j
    incr = Input(shape=(m,))
    logprice= Lambda(lambda x : K.log(x))(price)
    logprice = Add()([logprice, incr])
    pricenew=Lambda(lambda x : K.exp(x))(logprice)# creating the price at time j+1
    priceincr=Subtract()([pricenew, price])
    hedgenew = Multiply()([strategy, priceincr])
    hedge = Add()([hedge,hedgenew]) # building up the discretized stochastic integral
    inputs = inputs + [incr]
    price=pricenew
payoff = Lambda(lambda x : K.abs(x-strike) * pow(math.e, -r * T) - priceBS )(price)
outputs = Subtract()([payoff,hedge]) # payoff minus price minus hedge

inputs = inputs
outputs= outputs

model_hedge = Model(inputs=inputs, outputs=outputs)
```

I added r into the formula here.

$$
\begin{aligned}
d\log S &= \frac{d\log S}{dS}dS + \frac{1}{2}\sigma^2 S^2 \frac{d^2\log S}{dS^2}dt \\
&= \frac{1}{S}(\mu S dt + \sigma S dW) - \frac{1}{2}\sigma^2 dt \\
&= \left(\mu - \frac{\sigma^2}{2}\right) dt + \sigma dW
\end{aligned}
$$

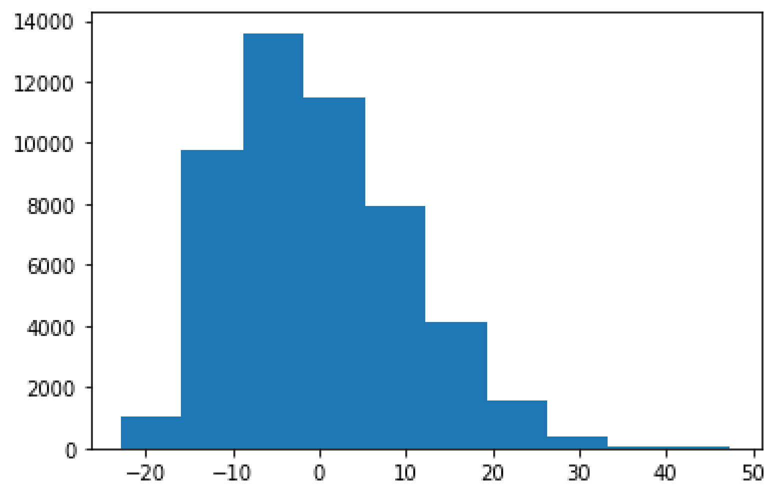
In [175]:
```
Ktrain = 5*10**4
initialprice = S0

# xtrain = [the price S0] + [the initial hedging being 0] + [the increments of the log price process]
xtrain = ([initialprice*np.ones((Ktrain,m))] +
          [np.zeros((Ktrain,m))]+
          [np.random.normal((r - (sigma ** 2 / 2)) * (T / N),sigma*np.sqrt(T)/np.sqrt(N),(Ktrain,m)) for i in range(N)])

# ytrain = output of hedging error initiated at 0
ytrain=np.zeros((Ktrain,1))
```

In [176]:
```
model_hedge.compile(optimizer='adam',loss='mean_squared_error')
```

In [177]:

```
import matplotlib.pyplot as plt

trajectory = []

for i in range(10):
    model_hedge.fit(x=xtrain,y=ytrain, epochs=1,verbose=False)
    weights = model_hedge.get_weights()
    trajectory = trajectory + weights
    plt.hist(model_hedge.predict(xtrain))
    plt.show()
    print(np.mean(model_hedge.predict(xtrain)))
```

```
-0.122332565
```




```
-0.12038415
```

In [178]:

```
weights = model_hedge.get_weights()
```

In [179]:

```
#This works when the number of layers equals d=2

def deltastrategy(s,j):
    length=s.shape[0]
    g=np.zeros(length)
    for p in range(length):
        ghelper=np.tanh(s[p]*(weights[j*2*d])+weights[j*2*d+1])
        #for k in range(1,d-1): #this line has to be checked in the case for d >2
        #    ghelper=np.tanh(np.matmul(weights[2*k+j*2*d], ghelper[0])+weights[2*k+j*2*d+1])
        g[p]=np.sum(np.squeeze(weights[2*(d-1)+j*2*d])*np.squeeze(ghelper))
        g[p]=g[p]+weights[2*d-1+j*2*d]
    return g
```

In [180]:

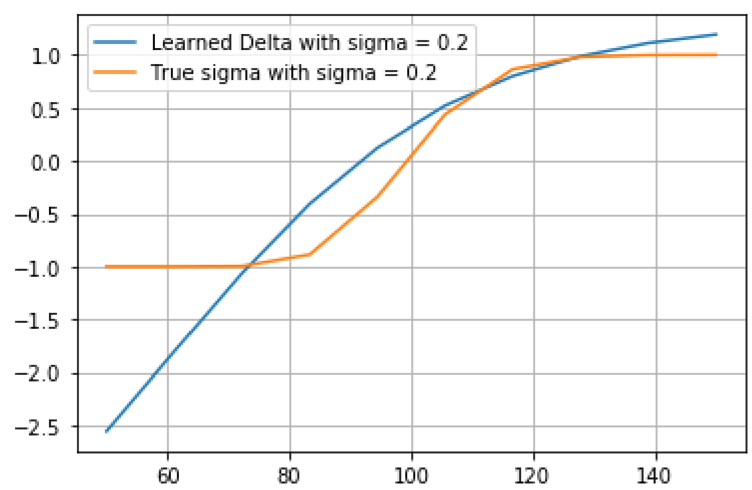
```
s=np.linspace(50,150,10) # range for the asset price to compute the hedging strategy
k=21 # choose k between 1 and N-1

learneddelta=deltastrategy(s,k)
learneddelta
```

Out[180]:

```
array([-2.55632424, -1.80474317, -1.0625881 , -0.40569887,  0.12380806,
        0.51947105,  0.79864633,  0.98774946,  1.11233413,  1.19292676])
```

In [181]:

```
# This plots the true versus the learned Hedging strategy
plt.plot(s,learneddelta,s,true_delta)
plt.grid(True)
plt.legend(["Learned Delta with sigma = 0.2","True sigma with sigma = 0.2"])
plt.show()
```



Problem(c)

Repeat (a) and (b) when the stock volatility is σ = 0.5.

Straddle

In [151]:
```
N=30 # time disrectization
S0=100 # initial value of the asset
strike=100 # strike for the call option
T=1.0 # maturity
sigma=0.5 # volatility in Black Scholes
r = 0.01 # interest rates
R=10 # number of Trajectories

s=np.linspace(50,150,10) # range for the asset price to compute the hedging strategy
k=21 # choose k between 1 and N-1

truedelta_call = delta_call(N,s,strike,T,sigma,r, k)
truedelta_put = delta_put(N,s,strike,T,sigma,r, k)
true_delta = truedelta_call + truedelta_put
```

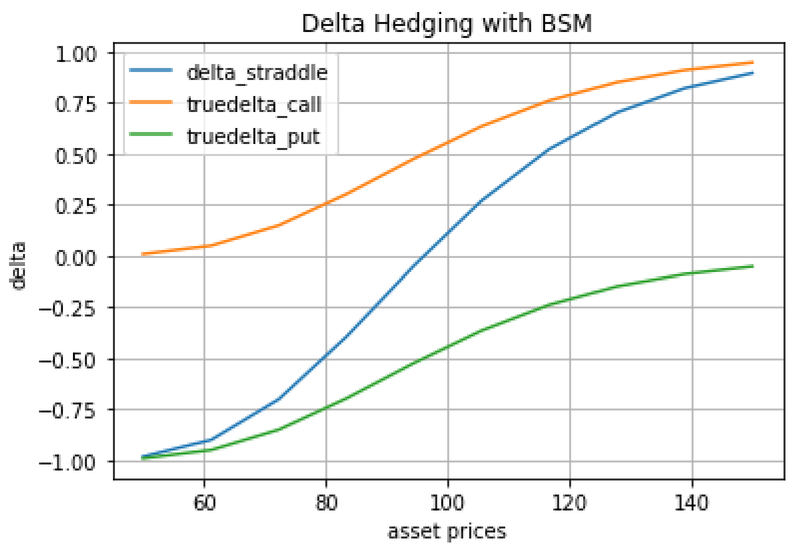
In [152]:
```
import pandas as pd
table = pd.DataFrame({"Asset Price" : s, "Call" : truedelta_call,"Put" : truedelta_put, "Straddle" : true_delta})
table
```

Out[152]:

	Asset Price	Call	Put	Straddle
0	50.000000	0.008583	-0.991417	-0.982834
1	61.111111	0.049432	-0.950568	-0.901136
2	72.222222	0.149079	-0.850921	-0.701841
3	83.333333	0.302278	-0.697722	-0.395443
4	94.444444	0.475748	-0.524252	-0.048504
5	105.555556	0.635070	-0.364930	0.270139
6	116.666667	0.761385	-0.238615	0.522769
7	127.777778	0.851513	-0.148487	0.703027
8	138.888889	0.911076	-0.088924	0.822153
9	150.000000	0.948284	-0.051716	0.896567

Delta curve becomes flatter as sigma goes up.

In [153]:
```
plt.plot(s,true_delta,s,truedelta_call,s,truedelta_put)
plt.xlabel('asset prices')
plt.ylabel("delta")
plt.title("Delta Hedging with BSM")
plt.grid(True)
plt.legend(["delta_straddle","truedelta_call","truedelta_put"])
plt.show()
```



Deep Hedging

In [154]:
```
priceBS1=BSM_call(S0,strike,sigma,T,r)
priceBS2=BSM_put(S0,strike,sigma,T,r)
priceBS = priceBS1 + priceBS2
print('Price of a Call option in the Black scholes model with initial price', S0, 'strike', strike, 'maturity', T , 'and volatility' ,
```

```
Price of a Call option in the Black scholes model with initial price 100 strike 100 maturity 1.0 and volatility 0.5 is equal to 39.
29379595463704
```

In [155]:
```
#Definition of neural networks for hedging strategies

m = 1 # dimension of price
d = 2 # number of layers in strategy
n = 32  # nodes in the first but last layers

# architecture is the same for all networks
layers = []
for j in range(N): # a neural network for each time step
    for i in range(d):
        if i < d-1:
            nodes = n
            layer = Dense(nodes, activation='tanh',trainable=True,
                      kernel_initializer=initializers.RandomNormal(0,1),#kernel_initializer='random_normal',
                      bias_initializer='random_normal',
                      name=str(i)+str(j))
        else:
            nodes = m
            layer = Dense(nodes, activation='linear', trainable=True,
                          kernel_initializer=initializers.RandomNormal(0,1),#kernel_initializer='random_normal',
                          bias_initializer='random_normal',
                          name=str(i)+str(j))
        layers = layers + [layer]
```

For the payoff, we should use (-r * T) to discount!

In [156]:
```
#Implementing the loss function
# Inputs is the training set below, containing the price S0,
#the initial hedging being 0, and the increments of the log price process
price = Input(shape=(m,))
hedge = Input(shape=(m,))

inputs = [price]+[hedge]


for j in range(N):
    strategy = price
    for k in range(d):
        strategy= layers[k+(j)*d](strategy) # hedging strategy at j , i.e. the neural network g_j
    incr = Input(shape=(m,))
    logprice= Lambda(lambda x : K.log(x))(price)
    logprice = Add()([logprice, incr])
    pricenew=Lambda(lambda x : K.exp(x))(logprice)# creating the price at time j+1
    priceincr=Subtract()([pricenew, price])
    hedgenew = Multiply()([strategy, priceincr])
    hedge = Add()([hedge,hedgenew]) # building up the discretized stochastic integral
    inputs = inputs + [incr]
    price=pricenew
#payoff1= Lambda(lambda x : 0.5*(K.abs(x-strike)+x-strike) - priceBS1 )(price) * np.exp(-r * T)
#payoff2= Lambda(lambda x : 0.5*(K.abs(strike-x)+strike-x) - priceBS2 )(price) * np.exp(-r * T)
#payoff = payoff1 + payoff2
payoff = Lambda(lambda x : K.abs(x-strike) * pow(math.e, -r * T) - priceBS )(price)
outputs = Subtract()([payoff,hedge]) # payoff minus price minus hedge

inputs = inputs
outputs= outputs

model_hedge = Model(inputs=inputs, outputs=outputs)
```

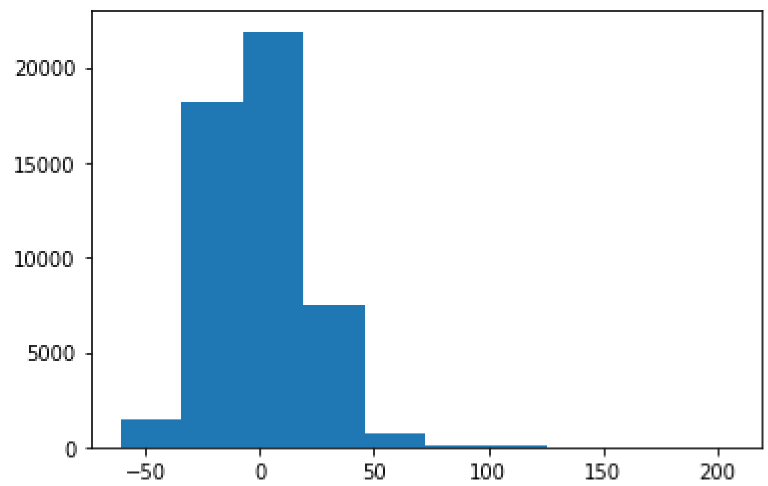
Add r here.

$$
\begin{aligned}
d\log S &= \frac{d\log S}{dS}dS + \frac{1}{2}\sigma^2 S^2 \frac{d^2\log S}{dS^2}dt \\
&= \frac{1}{S}(\mu S dt + \sigma S dW) - \frac{1}{2}\sigma^2 dt \\
&= \left(\mu - \frac{\sigma^2}{2}\right) dt + \sigma dW
\end{aligned}
$$

In [157]:
```
Ktrain = 5*10**4
initialprice = S0

# xtrain = [the price S0] + [the initial hedging being 0] + [the increments of the log price process]
xtrain = ([initialprice*np.ones((Ktrain,m))] +
          [np.zeros((Ktrain,m))]+
          [np.random.normal((r - (sigma ** 2 / 2)) * (T / N),sigma*np.sqrt(T)/np.sqrt(N),(Ktrain,m)) for i in range(N)])

# ytrain = output of hedging error initiated at 0
ytrain=np.zeros((Ktrain,1))
```

In [158]:
```
model_hedge.compile(optimizer='adam',loss='mean_squared_error')
```

In [159]:
```
import matplotlib.pyplot as plt

trajectory = []

for i in range(10):
    model_hedge.fit(x=xtrain,y=ytrain, epochs=1,verbose=False)
    weights = model_hedge.get_weights()
    trajectory = trajectory + weights
    plt.hist(model_hedge.predict(xtrain))
    plt.show()
    print(np.mean(model_hedge.predict(xtrain)))
```




```
-0.18729812
```

In [161]:
```
weights = model_hedge.get_weights()
```

In [162]:
```
#This works when the number of layers equals d=2

def deltastrategy(s,j):
    length=s.shape[0]
    g=np.zeros(length)
    for p in range(length):
        ghelper=np.tanh(s[p]*(weights[j*2*d])+weights[j*2*d+1])
        #for k in range(1,d-1): #this line has to be checked in the case for d >2
        #    ghelper=np.tanh(np.matmul(weights[2*k+j*2*d], ghelper[0])+weights[2*k+j*2*d+1])
        g[p]=np.sum(np.squeeze(weights[2*(d-1)+j*2*d])*np.squeeze(ghelper))
        g[p]=g[p]+weights[2*d-1+j*2*d]
    return g
```

In [163]:
```
s=np.linspace(50,150,10) # range for the asset price to compute the hedging strategy
k=21 # choose k between 1 and N-1

learneddelta=deltastrategy(s,k)
learneddelta
```

Out[163]:
```
array([-1.18426609, -0.96801877, -0.63164628, -0.28620753,  0.02079334,
        0.2743924 ,  0.47487953,  0.62886953,  0.74481726,  0.83090377])
```

We can see that as sigma goes up, the hedging becomes more and more precise!

In [164]:

```
# This plots the true versus the learned Hedging strategy
plt.plot(s,learneddelta,s,true_delta)
plt.grid(True)
plt.legend(["Learned Delta with sigma = 0.5","True sigma with sigma = 0.5"])
plt.show()
```