In [52]:

```
import os
import torch
from torch import nn
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt

from torch.utils.data import DataLoader
from torchvision import transforms, datasets
from torch.utils.data.sampler import SubsetRandomSampler
```

In [22]:

```
#Defining transform to convert the images
transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5),(0.5),)])
#Load the dataset
trainSet = datasets.FashionMNIST('~/.pytorch/F_MNIST_data',download=True, train=True, trans
testSet = datasets.FashionMNIST('~/.pytorch/F_MNIST_data',download=True, train=False, trans

#Preparing for validation test
indices = list(range(len(trainSet)))
np.random.shuffle(indices)
#Get 20%  for the test set
split = int(np.floor(0.2 * len(trainSet)))
trainSample =  SubsetRandomSampler(indices[:split])
validSample =  SubsetRandomSampler(indices[split:])

#Data Loader
trainLoader = torch.utils.data.DataLoader(trainSet, sampler=trainSample, batch_size=32)
validLoader = torch.utils.data.DataLoader(trainSet, sampler=validSample, batch_size=32)
testLoader = torch.utils.data.DataLoader(testSet, batch_size=64, shuffle=True)
```

In [110]:

```

# define NN architecture
class MLP(nn.Module):
    def __init__(self):
        super(MLP,self).__init__()

        hidden_1 = 512
        hidden_2 = 512

        self.fc1 = nn.Linear(28*28, 512)

        self.fc2 = nn.Linear(512,512)

        self.fc3 = nn.Linear(512,512)

        self.fc4 = nn.Linear(512,512)

        self.fc5 = nn.Linear(512,10)
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.droput = nn.Dropout(0.2)

    def forward(self,x):
        # flatten image input
        x = x.view(-1,28*28)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.droput(x)
         # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.droput(x)
        x = F.relu(self.fc3(x))
        # add dropout layer
        x = self.droput(x)
        x = F.relu(self.fc4(x))
        # add dropout layer
        x = self.droput(x)
        # add output layer
        x = F.softmax(self.fc3(x))
        return x
```

In [111]:

```
# Initialize the MLP
mlp = MLP()

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp.parameters(), lr=0.01)
```

In [112]:

```
# number of epochs to train the model
n_epochs = 20
train_losses = []
valid_losses = []
# initialize tracker for minimum validation loss
valid_loss_min = np.Inf  # set initial "min" to infinity
for epoch in range(n_epochs):
    # monitor losses
    train_loss = 0
    valid_loss = 0


    ###################
    # train the model #
    ###################
    mlp.train() # prep model for training
    for data,label in trainLoader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = mlp(data)
        # calculate the loss
        loss = criterion(output,label)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item() * data.size(0)


     ######################
    # validate the model #
    ######################
    mlp.eval()  # prep model for evaluation
    for data,label in validLoader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = mlp(data)
        # calculate the loss
        loss = criterion(output,label)
        # update running validation loss
        valid_loss = loss.item() * data.size(0)

    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss = train_loss / len(trainLoader.sampler)
    valid_loss = valid_loss / len(validLoader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss
        ))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(mlp.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

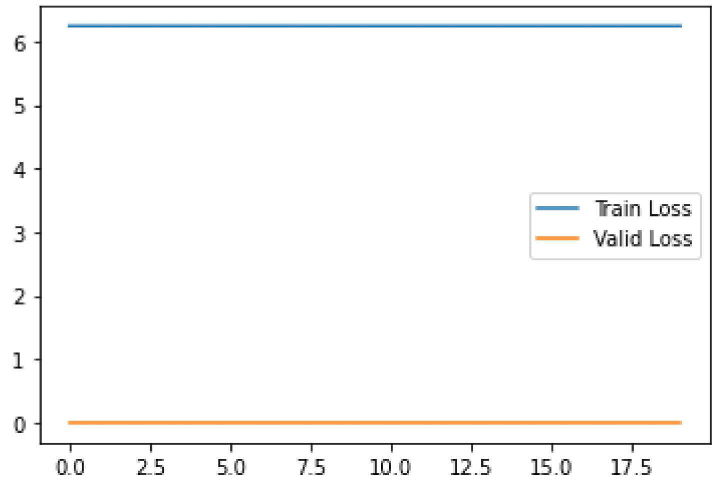
```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:40: UserWarnin
g: Implicit dimension choice for softmax has been deprecated. Change the cal
l to include dim=X as an argument.

Epoch: 1 	Training Loss: 6.238286 	Validation Loss: 0.004159
Validation loss decreased (inf --> 0.004159).  Saving model ...
Epoch: 2 	Training Loss: 6.238284 	Validation Loss: 0.004159
Epoch: 3 	Training Loss: 6.238281 	Validation Loss: 0.004159
Epoch: 4 	Training Loss: 6.238279 	Validation Loss: 0.004159
Epoch: 5 	Training Loss: 6.238276 	Validation Loss: 0.004159
Epoch: 6 	Training Loss: 6.238274 	Validation Loss: 0.004159
Epoch: 7 	Training Loss: 6.238271 	Validation Loss: 0.004159
Validation loss decreased (0.004159 --> 0.004159).  Saving model ...
Epoch: 8 	Training Loss: 6.238268 	Validation Loss: 0.004159
Validation loss decreased (0.004159 --> 0.004159).  Saving model ...
Epoch: 9 	Training Loss: 6.238266 	Validation Loss: 0.004159
Epoch: 10 	Training Loss: 6.238263 	Validation Loss: 0.004159
Epoch: 11 	Training Loss: 6.238260 	Validation Loss: 0.004159
Validation loss decreased (0.004159 --> 0.004159).  Saving model ...
Epoch: 12 	Training Loss: 6.238257 	Validation Loss: 0.004159
Epoch: 13 	Training Loss: 6.238255 	Validation Loss: 0.004159
Epoch: 14 	Training Loss: 6.238251 	Validation Loss: 0.004159
Epoch: 15 	Training Loss: 6.238249 	Validation Loss: 0.004159
Validation loss decreased (0.004159 --> 0.004159).  Saving model ...
Epoch: 16 	Training Loss: 6.238245 	Validation Loss: 0.004159
Validation loss decreased (0.004159 --> 0.004159).  Saving model ...
Epoch: 17 	Training Loss: 6.238243 	Validation Loss: 0.004159
Epoch: 18 	Training Loss: 6.238241 	Validation Loss: 0.004159
Validation loss decreased (0.004159 --> 0.004159).  Saving model ...
Epoch: 19 	Training Loss: 6.238237 	Validation Loss: 0.004159
Validation loss decreased (0.004159 --> 0.004159).  Saving model ...
Epoch: 20 	Training Loss: 6.238234 	Validation Loss: 0.004159
```

In [113]:

```
plt.plot(train_losses, label ='Train Loss')
plt.plot(valid_losses, label ='Valid Loss')
plt.legend()
```

Out[113]:

```
<matplotlib.legend.Legend at 0x7fe2aca9b250>
```



A. Note the number of trainable parameters for this network.

No. of parameters are: 1195018.

In [117]:

```
#number of trainable parameters for this network
print(sum(p.numel() for p in mlp.parameters() if p.requires_grad))
```

```
1195018
```

In [118]:

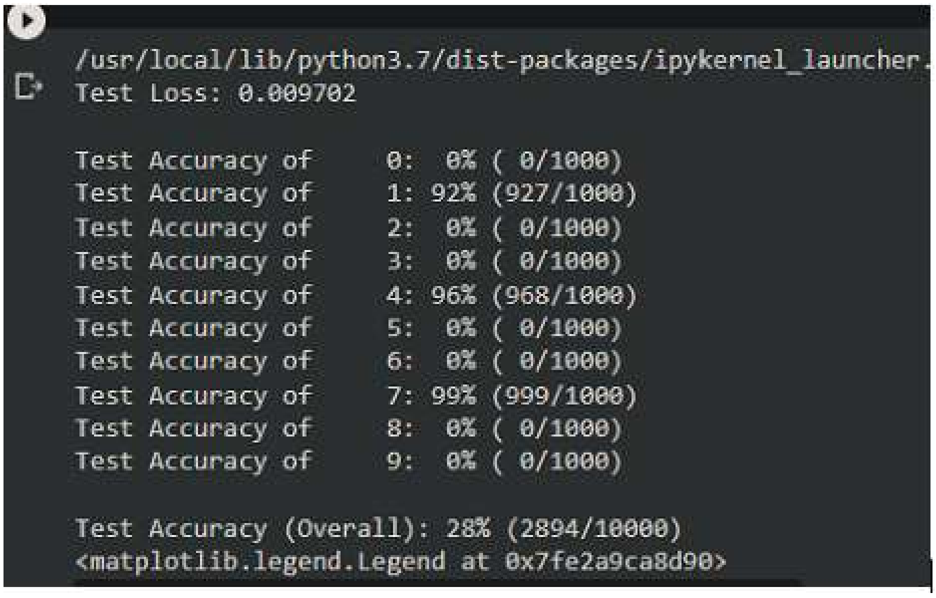
```
#Load the Model with Lowest Validation Loss
mlp.load_state_dict(torch.load('model.pt'))
```

Out[118]:

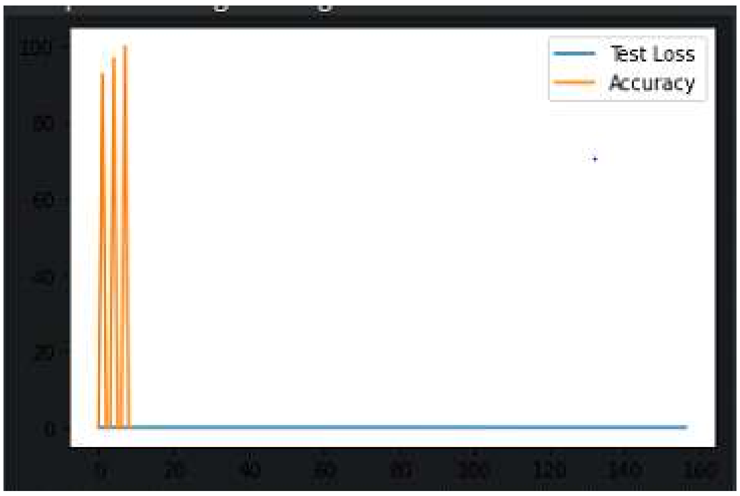
```
<All keys matched successfully>
```

B. Describe the accuracy you obtain and show the loss/ accuracy curves.

Accuracy obtain was 28% before applying dropout layers.



```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.
Test Loss: 0.009702

Test Accuracy of     0:  0% ( 0/1000)
Test Accuracy of     1: 92% (927/1000)
Test Accuracy of     2:  0% ( 0/1000)
Test Accuracy of     3:  0% ( 0/1000)
Test Accuracy of     4: 96% (968/1000)
Test Accuracy of     5:  0% ( 0/1000)
Test Accuracy of     6:  0% ( 0/1000)
Test Accuracy of     7: 99% (999/1000)
Test Accuracy of     8:  0% ( 0/1000)
Test Accuracy of     9:  0% ( 0/1000)

Test Accuracy (Overall): 28% (2894/10000)
<matplotlib.legend.Legend at 0x7fe2a9ca8d90>
```



Accuracy obtain is 28% and the loss/ accuracy curves are below:

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

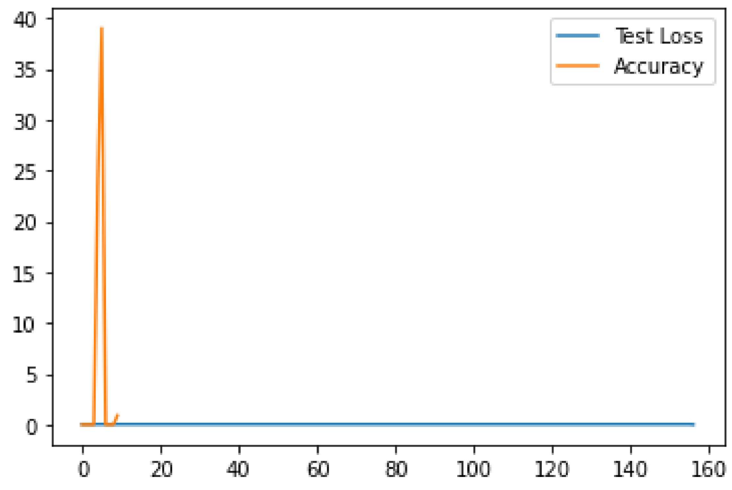
In [119]:

```
#Test the Trained Model
# initialize lists to monitor test loss and accuracy
test_loss = 0.0
test_losses = []
accuracy = []
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
mlp.eval() # prep model for evaluation
for data, target in testLoader:
    # forward pass: compute predicted outputs by passing inputs to the model
    output = mlp(data)
    # calculate the loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct = np.squeeze(pred.eq(target.data.view_as(pred)))
    # calculate test accuracy for each object class
    for i in range(len(target)):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1
    test_loss = test_loss/len(testLoader.sampler)
    test_losses.append(test_loss)
# calculate and print avg test loss

print('Test Loss: {:.6f}\n'.format(test_loss))
for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            str(i), 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
        accuracy.append(100 * class_correct[i] / class_total[i])
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))

plt.plot(test_losses, label = 'Test Loss')
plt.plot(accuracy, label = 'Accuracy')
plt.legend()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:40: UserWarnin
g: Implicit dimension choice for softmax has been deprecated. Change the cal
l to include dim=X as an argument.

Test Loss: 0.009985

Test Accuracy of     0:  0% ( 0/1000)
Test Accuracy of     1:  0% ( 0/1000)
Test Accuracy of     2:  0% ( 0/1000)
Test Accuracy of     3:  0% ( 0/1000)
Test Accuracy of     4: 24% (241/1000)
Test Accuracy of     5: 39% (390/1000)
Test Accuracy of     6:  0% ( 0/1000)
Test Accuracy of     7:  0% ( 0/1000)
Test Accuracy of     8:  0% ( 0/1000)
Test Accuracy of     9:  0% ( 9/1000)

Test Accuracy (Overall):  6% (640/10000)
```

Out[119]:

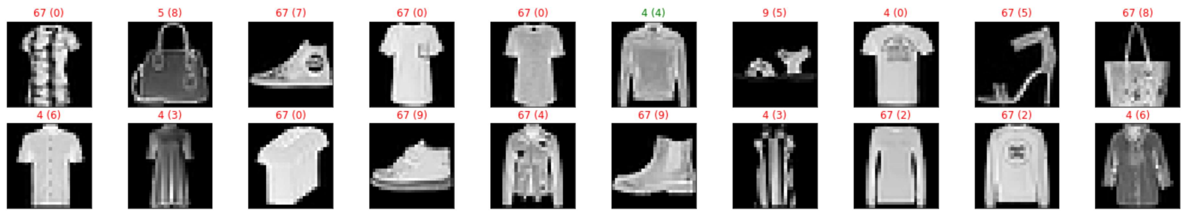
```
<matplotlib.legend.Legend at 0x7fe2aca14cd0>
```



In [120]:

```python
#Visualize the test Result
# obtain one batch of test images
dataiter = iter(testLoader)
images, labels = dataiter.next()
# get sample outputs
output = mlp(images)
# convert output probabilities to predicted class
_, preds = torch.max(output, 1)
# prep images for display
images = images.numpy()
# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(images[idx]), cmap='gray')
    ax.set_title("{} ({})".format(str(preds[idx].item()), str(labels[idx].item())),
                 color=("green" if preds[idx]==labels[idx] else "red"))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:40: UserWarnin
g: Implicit dimension choice for softmax has been deprecated. Change the cal
l to include dim=X as an argument.
```

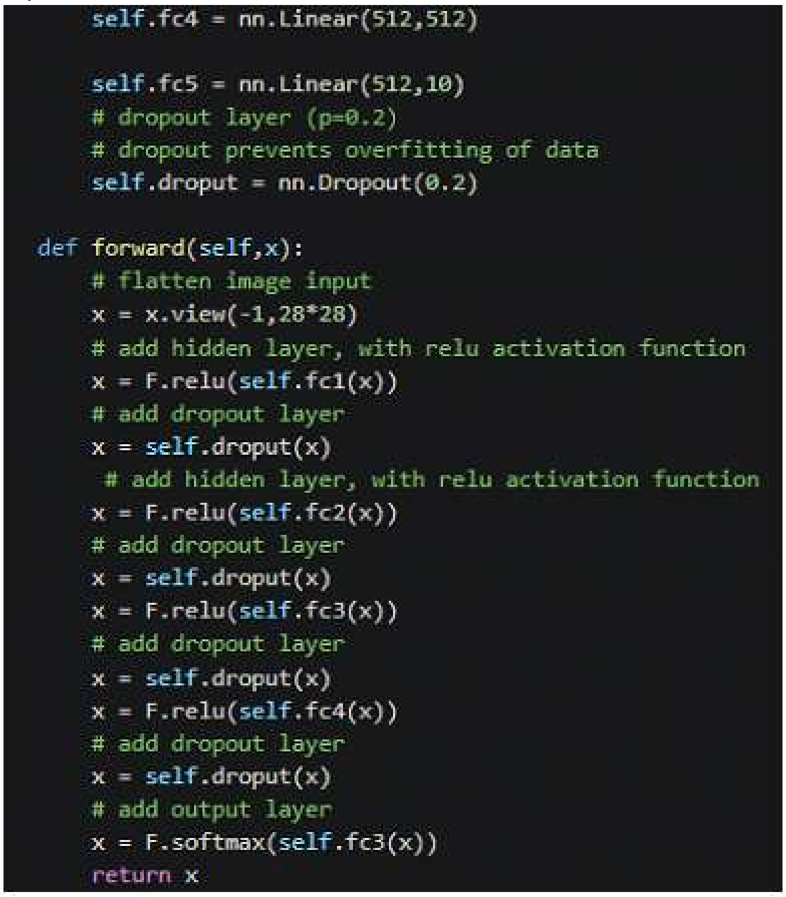


C. Keep on training the network, do you observe any signs of overfitting — describe your observations? Implement the use of regularization using dropout on

the network above—change the dropout parameter to see its effect? Document and explain the results that you obtain.

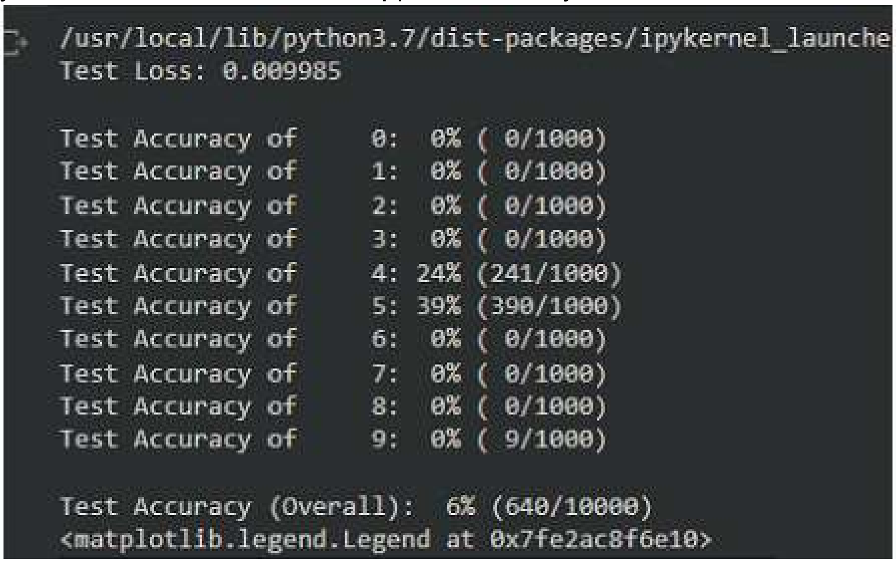
Yes, I observe overfitting in the network because our model has remembered the images and can't make decision when a random image came, in short it has no ability to classify any image which it has not seen before.

Implementing the dropout parameter:



```
        self.fc4 = nn.Linear(512,512)

        self.fc5 = nn.Linear(512,10)
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.droput = nn.Dropout(0.2)

    def forward(self,x):
        # flatten image input
        x = x.view(-1,28*28)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.droput(x)
         # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.droput(x)
        x = F.relu(self.fc3(x))
        # add dropout layer
        x = self.droput(x)
        x = F.relu(self.fc4(x))
        # add dropout layer
        x = self.droput(x)
        # add output layer
        x = F.softmax(self.fc3(x))
        return x
```

Here is the Accuracy and it can be seen, it has dropped drastically from 28% to 6%.



```
/usr/local/lib/python3.7/dist-packages/ipykernel_launche
Test Loss: 0.009985

Test Accuracy of     0:  0% ( 0/1000)
Test Accuracy of     1:  0% ( 0/1000)
Test Accuracy of     2:  0% ( 0/1000)
Test Accuracy of     3:  0% ( 0/1000)
Test Accuracy of     4: 24% (241/1000)
Test Accuracy of     5: 39% (390/1000)
Test Accuracy of     6:  0% ( 0/1000)
Test Accuracy of     7:  0% ( 0/1000)
Test Accuracy of     8:  0% ( 0/1000)
Test Accuracy of     9:  0% ( 9/1000)

Test Accuracy (Overall):  6% (640/10000)
<matplotlib.legend.Legend at 0x7fe2ac8f6e10>
```

D. Visualize the performance of the model learned by

putting the model in evaluation mode and giving it input and see what outputs are produced. Show 5 sample images and the corresponding outputs from the neural network for those input images.

Hence in the study it is proved as well that MLP doesn’t perform well as image classifier.

Type *Markdown* and LaTeX: α^2