Name: Ziyang Zhang

UNI: zz2732

In [234]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# sklearn
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, cross_val_score, L
eaveOneOut, KFold
from sklearn import svm
from sklearn import metrics
from sklearn import preprocessing

# PyTorch
import torch
import torchvision
from torchvision import transforms
from torch.utils import data
from torch import nn
import torch.nn.functional as F

# Image
from IPython.display import Image

%matplotlib inline
sns.set_style("darkgrid")
```

Question 1 SVMs

1.1 Scaling the inputs.

True. It is generally a good idea to scale all input variables.
Because in SVM, we use Kernel tricks to measure the similarity between feature vectors, and all Kernels are measuring the distances(in different ways) between feature vectors. Without scaling all features into similar ranges, the features that have largest range will be most likely to dominate the Kernel function and disregard the relative differences, therefore making the model unable to learn from other important features as expected.
For example, if the i^{th} column of the feature vector has a larger scale than all other components, the corresponding coefficient i in the optimial solution will be smaller than the other components. But it doesn't mean the i^{th} feature is actually less important. It is only due to its much larger scale. Therefore, we should scale all the inputs to avoid this issue.

1.2 Classifying Tumors

1.2.1 Build a SVM classifier using a 'linear' kernel, test performance using Cross-Validation

In [96]:
```
# Load the dataset.
data=load_breast_cancer()
print(data.data.shape)
print(data.feature_names)
print(data.target.shape)
print(data.target_names)
```

```
(569, 30)
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
(569,)
['malignant' 'benign']
```

In [97]:
```
# Standardize the features
scaler=preprocessing.StandardScaler().fit(data.data)
X=scaler.transform(data.data)
Y=data.target
```

In [98]:
```
# t=70%
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3)

#Create a svm Classifier
clf = svm.SVC(kernel='linear') # Linear Kernel

#Train the model using the training sets
clf.fit(X_train, y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)
```

In [99]:
```
# accuracy
clf.score(X_test,y_test)
```

Out[99]:
```
0.9649122807017544
```

In [100]:
```
# cross-validation test set performance
# LINEAR Kernel
clf = svm.SVC(kernel='linear')
scores = cross_val_score(clf, X, Y, cv=5)
scores
```

Out[100]:
```
array([0.95614035, 0.98245614, 0.96491228, 0.96491228, 0.98230088])
```

In [101]:
```
# Mean with 95% Confidence Interval of Accuracy
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
Accuracy: 0.97 (+/- 0.02)
```

In [102]:
```
# cross-validation test set performance
# POLY Kernel
clf = svm.SVC(kernel='poly')
scores = cross_val_score(clf, X, Y, cv=5)
scores
```

Out[102]:
```
array([0.87719298, 0.87719298, 0.89473684, 0.90350877, 0.9380531 ])
```

In [103]:
```
# Mean with 95% Confidence Interval of Accuracy
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
Accuracy: 0.90 (+/- 0.04)
```

In [104]:
```
# cross-validation test set performance
# RBF Kernel
clf = svm.SVC(kernel='rbf')
scores = cross_val_score(clf, X, Y, cv=5)
scores
```

Out[104]:
```
array([0.97368421, 0.95614035, 1.        , 0.96491228, 0.97345133])
```

In [105]:
```
# Mean with 95% Confidence Interval of Accuracy
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
Accuracy: 0.97 (+/- 0.03)
```

Conclusion:

Choose the SVM classifier with 'Linear' Kernel, since it has the highest mean accuracy with a slightly smaller 95% confidence interval.

1.2.2 Repeat 50 times

In [106]:
```
scores_50=np.array([])
for i in range(50):
    # t=70%
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=
0.3)

    #Create a svm Classifier
    clf = svm.SVC(kernel='linear') # Linear Kernel

    #Train the model using the training sets
    clf.fit(X_train, y_train)

    #Predict the response for test dataset
    y_pred = clf.predict(X_test)

    scores_50=np.append(scores_50,clf.score(X_test,y_test))
scores_50
```

Out[106]:
```
array([0.97660819, 0.97660819, 0.96491228, 0.95906433, 0.98830409,
       0.96491228, 0.95906433, 0.95321637, 0.96491228, 0.94152047,
       0.98245614, 0.98245614, 0.98830409, 0.97660819, 0.97076023,
       0.97660819, 0.97076023, 0.95321637, 0.96491228, 0.98245614,
       0.95906433, 0.96491228, 0.97660819, 0.98830409, 0.97076023,
       0.94736842, 1.        , 0.96491228, 0.98245614, 0.96491228,
       0.97660819, 0.97076023, 0.98830409, 0.94736842, 0.97076023,
       0.98245614, 0.95906433, 0.98830409, 0.97660819, 0.98830409,
       0.96491228, 0.98245614, 0.98245614, 0.94736842, 0.97660819,
       0.96491228, 0.97660819, 0.95906433, 0.97660819, 0.98830409])
```

In [107]:
```
# Mean and Standard Deviation of test-set performances
print("Accuracy: %0.2f, Standard Deviation: %0.2f)" % (scores_50.mean(),
scores_50.std()))
```

```
Accuracy: 0.97, Standard Deviation: 0.01)
```

1.2.3 Repeat (b) for values of t=50%,55%,...,95%, t is the percentage of the data assigned into training set.

In [108]:
```
# compute the % for the test data set size
t_range=np.arange(0.5,1,0.05)
t_test=1-t_range
t_test
```

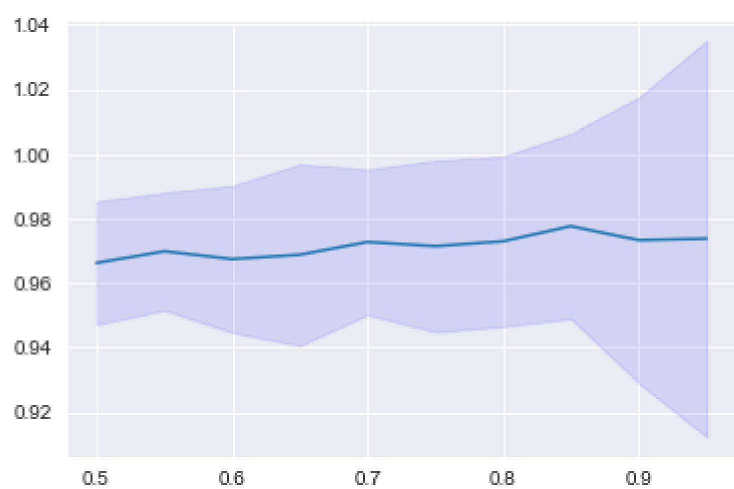
Out[108]:
```
array([0.5 , 0.45, 0.4 , 0.35, 0.3 , 0.25, 0.2 , 0.15, 0.1 , 0.05])
```

In [135]:
```
mean=np.array([])
ci_95=np.array([])
for t in t_test:
    scores_50=np.array([])
    for i in range(50):
        X_train, X_test, y_train, y_test = train_test_split(X, Y, test_s
ize=t)

        #Create a svm Classifier
        clf = svm.SVC(kernel='linear') # Linear Kernel

        #Train the model using the training sets
        clf.fit(X_train, y_train)

        #Predict the response for test dataset
        y_pred = clf.predict(X_test)

        scores_50=np.append(scores_50,clf.score(X_test,y_test))
    mean=np.append(mean,scores_50.mean())
    #ci=1.96 * np.sqrt( (scores_50.mean() * (1 - scores_50.mean())) / 5
0)
    ci=1.96 * np.std(scores_50)/np.mean(scores_50)
    ci_95=np.append(ci_95,ci)
print(mean)
print(ci_95)
```

```
[0.96624561 0.96988327 0.96745614 0.9688     0.97274854 0.97146853
 0.97298246 0.97767442 0.97333333 0.9737931 ]
[0.01909225 0.01825974 0.02276465 0.02814685 0.02255593 0.02659657
 0.02640881 0.02867933 0.04426028 0.06139115]
```

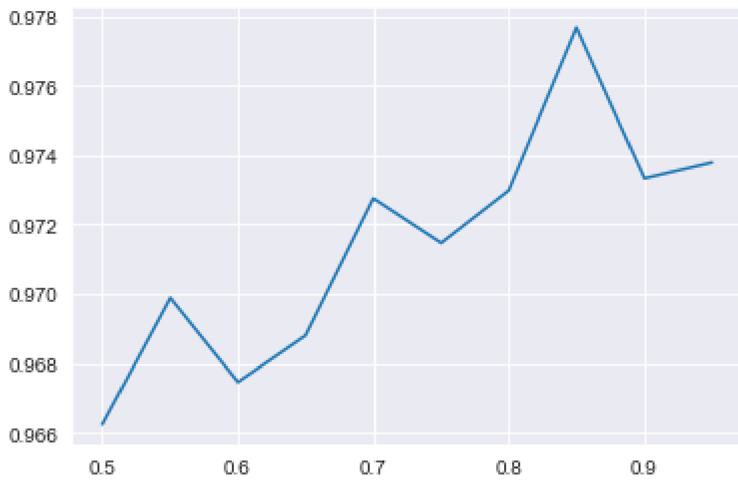
In [136]:
```
# plot of t and mean test set performance with confidence intervals
fig, ax = plt.subplots()
ax.plot(t_range,mean)
ax.fill_between(t_range, (mean-ci_95), (mean+ci_95), color='b', alpha=.1
)
```

Out[136]: <matplotlib.collections.PolyCollection at 0x7fa965776fd0>



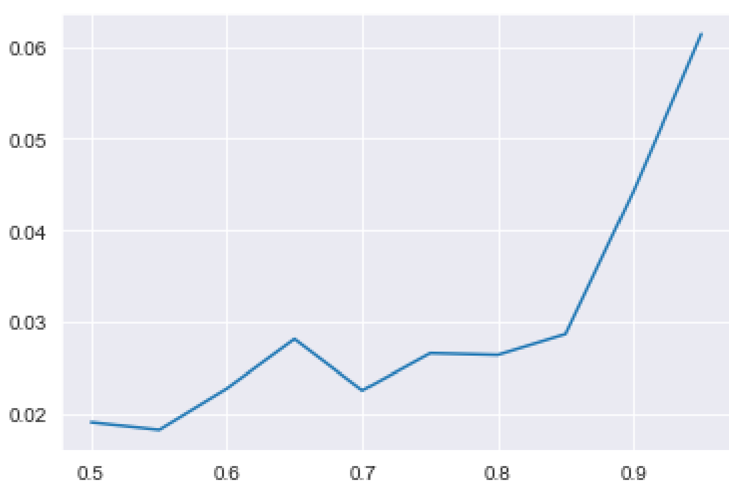
In [137]:
```
# plot of t and mean test set performances
plt.plot(t_range,mean)
```

Out[137]: [<matplotlib.lines.Line2D at 0x7fa965b0f7d0>]



In [138]:
```
# plot of t and 95% C.I.
plt.plot(t_range,ci_95)
```

Out[138]: [<matplotlib.lines.Line2D at 0x7fa965b6f350>]



Conclusion:

- The accuracy first increases as we increase t, it makes sense because we have more and more data/information to train the classifier.

- However, at some point, the accuracy performance started decreasing, since we now have too little data to test the trained classifier, leading to overfitting using too much training data.

- The optimal % for the training data set from the plot above is around 0.8 with the highest test-set performance. The +/- margin for the 95% confidence interval gets wider and wider for all mean accuracies.

1.3 SVMs and Cross-Validation

Yes, there is potentially a problem with this approach, and we should not simply use the previously selected hyperparameters C and σ found in the first 10000 points.

Basically, by doing so, we are using too few data points to train the model in order to find the optimal hyperparameters C and σ. This will cause major problems especially when the 10000 data points are not representative enough for the whole dataset. Then the test accuracy would be super low. For example, if the model is timing sensitive of the training data, then using parameters from the old 10000 data points would do poorly on the test dataset from the newly added data points.

On the other scenario, even if the 10000 data points are completely randomly selected from the whole data set, the random noise for the hyperparameters C and σ estimates would still be relatively large, because the estimation was based on a much smaller dataset compared to the whole dataset.

Question 2 PyTorch Practice

2.1 Install PyTorch

In [ ]:
```
# conda install pytorch torchvision -c pytorch
```

2.2 Read through 60-minute blitz tutorial

Link: [Tutorial (https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)](https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)

2.3 Run some commands from the tutorial

Tensor basics

In [151]:
```
x=torch.empty(2,2, dtype=torch.float64)
x
```

Out[151]:
```
tensor([[-2.0000e+00, -2.0000e+00],
        [5.8119e+294, 7.2929e-304]], dtype=torch.float64)
```

In [152]:
```
x.size()
```
Out[152]:
```
torch.Size([2, 2])
```

In [153]:
```
x=torch.tensor([2.5,0.1,3.0])
x
```
Out[153]:
```
tensor([2.5000, 0.1000, 3.0000])
```

In [154]:
```
x=torch.rand(2,2)
y=torch.rand(2,2)
z=x+y   # elementwise addition
z
```
Out[154]:
```
tensor([[0.5390, 1.0577],
        [0.8645, 0.8174]])
```

In [155]:
```
z=torch.add(x,y)
z
```
Out[155]:
```
tensor([[0.5390, 1.0577],
        [0.8645, 0.8174]])
```

In [156]:
```
print(x)
print(y)
# in-place addition
y.add_(x)
y
```
```
tensor([[0.2212, 0.1353],
        [0.6998, 0.2694]])
tensor([[0.3178, 0.9224],
        [0.1647, 0.5480]])
```
Out[156]:
```
tensor([[0.5390, 1.0577],
        [0.8645, 0.8174]])
```

In [157]:
```
# indexing
x=torch.rand(5,3)
print(x)
print(x[:,1])
print(x[1,:])
```
```
tensor([[0.0383, 0.1965, 0.8128],
        [0.5330, 0.7704, 0.1896],
        [0.5458, 0.7511, 0.6874],
        [0.0520, 0.7792, 0.4089],
        [0.4288, 0.6361, 0.5694]])
tensor([0.1965, 0.7704, 0.7511, 0.7792, 0.6361])
tensor([0.5330, 0.7704, 0.1896])
```

Autograd

If you set its attribute `.requires_grad` as `True`, it starts to track all operations on it. When you finish your computation you can call `.backward()` and have all the gradients computed automatically. The gradient for this tensor will be accumulated into `.grad` attribute.

In [168]:
```
x = torch.ones(2, 2, requires_grad=True)  # start tracking all operation
s on 'x'
print(x)
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

In [169]:
```
y = x + 2
print(y)
```

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

In [170]:
```
print(y.grad_fn)
```

```
<AddBackward0 object at 0x7fa967f8c1d0>
```

In [171]:
```
z = y * y * 3
out = z.mean()
print(z)
print(out)
```

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>)
tensor(27., grad_fn=<MeanBackward0>)
```

In [173]:
```
out.backward()                # call '.backward()' to compute the gradients
```

In [174]:
```
x.grad                        # gradients are saved in the '.grad' attribut
e
```

Out[174]:
```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

In [175]:
```
# an example of vector-Jacobian product:
x = torch.randn(3, requires_grad=True)

y = x * 2
while y.data.norm() < 1000:
    y = y * 2

print(y)
```

```
tensor([  509.3170, -1573.0967,  -719.0090], grad_fn=<MulBackward0>)
```

Now in this case `y` is no longer a scalar. `torch.autograd` could not compute the full Jacobian directly, but if we just want the vector-Jacobian product, simply pass the vector to `backward` as argument:

In [176]:
```
v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
y.backward(v)

print(x.grad)
```

```
tensor([1.0240e+02, 1.0240e+03, 1.0240e-01])
```

Stop autograd from tracking

In [177]:
```
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```

```
True
True
False
```

Using `.detach()` to get a new tensor

In [178]:
```
print(x.requires_grad)
y = x.detach()
print(y.requires_grad)
print(x.eq(y).all())
```

```
True
False
tensor(True)
```

Neural Networks

In [237]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convoluti
on
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimensio
n
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

In [238]:
```
params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's .weight
```

```
10
torch.Size([6, 1, 3, 3])
```

In [239]:
```
input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

```
tensor([[ 0.0170,  0.0047,  0.0384,  0.1242,  0.1046,  0.0384,  0.1805,
0.0826,
          0.1323, -0.0334]], grad_fn=<AddmmBackward>)
```

In [240]:
```
net.zero_grad()
out.backward(torch.randn(1, 10))
```

In [241]:
```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

```
tensor(1.5707, grad_fn=<MseLossBackward>)
```

In [242]:
```
print(loss.grad_fn)  # MSELoss
print(loss.grad_fn.next_functions[0][0])  # Linear
print(loss.grad_fn.next_functions[0][0].next_functions[0][0])  # ReLU
```

```
<MseLossBackward object at 0x7fa94c27c7d0>
<AddmmBackward object at 0x7fa94c27ca10>
<AccumulateGrad object at 0x7fa94c27c7d0>
```

In [243]:
```
net.zero_grad()     # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([ 0.0219, -0.0151, -0.0204,  0.0258, -0.0018,  0.0282])
```

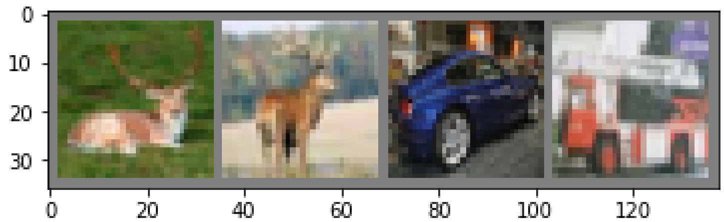
In [244]:
```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

Training a Classifier

In [3]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transfo
rm)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transfor
m)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz


Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

In [4]:
```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```




```
 deer  deer   car truck
```

In [5]:
```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

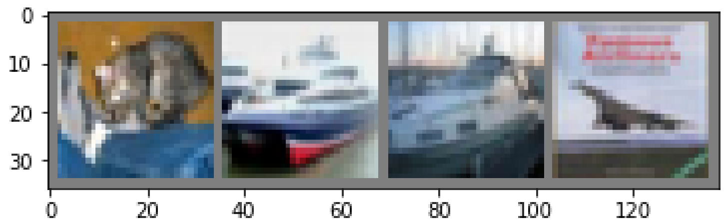
In [6]:
```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

In [7]:
```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.242
[1,  4000] loss: 1.879
[1,  6000] loss: 1.686
[1,  8000] loss: 1.589
[1, 10000] loss: 1.543
[1, 12000] loss: 1.480
[2,  2000] loss: 1.401
[2,  4000] loss: 1.384
[2,  6000] loss: 1.364
[2,  8000] loss: 1.329
[2, 10000] loss: 1.317
[2, 12000] loss: 1.313
Finished Training
```

In [8]:
```
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

In [9]:
```
dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in rang
e(4)))
```




```
GroundTruth:    cat  ship  ship plane
```

In [10]:
```
net = Net()
net.load_state_dict(torch.load(PATH))
```

Out[10]:
```
<All keys matched successfully>
```

In [11]:
```
outputs = net(images)
```

In [12]:
```
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

```
Predicted:    cat   car   car plane
```

In [13]:
```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 49 %
```

In [14]:
```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of plane : 51 %
Accuracy of   car : 60 %
Accuracy of  bird : 43 %
Accuracy of   cat : 44 %
Accuracy of  deer : 30 %
Accuracy of   dog : 23 %
Accuracy of  frog : 49 %
Accuracy of horse : 55 %
Accuracy of  ship : 68 %
Accuracy of truck : 72 %
```

2.4 Create a Neural Network with 2 ReLU hidden layers

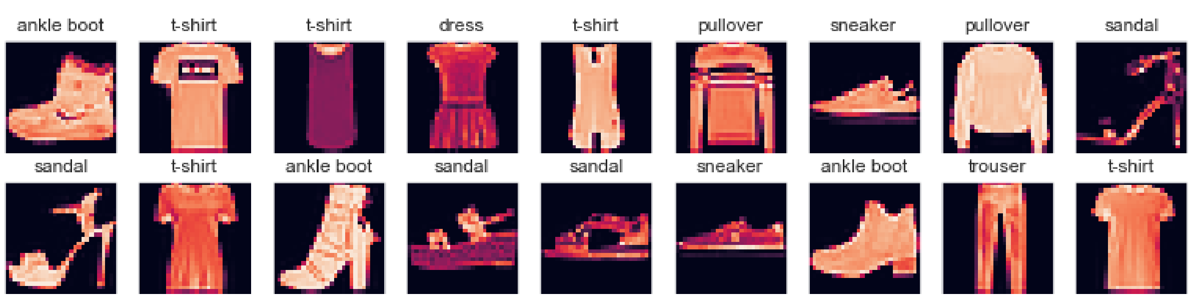
Load the Dataset: Fashio MNIST

In [183]:
```
trans = transforms.ToTensor()
mnist_train = torchvision.datasets.FashionMNIST(
    root="../../../datasets", train=True, transform=trans, download=Fals
e)
mnist_test = torchvision.datasets.FashionMNIST(
    root="../../../datasets", train=False, transform=trans, download=Fal
se)
len(mnist_train), len(mnist_test)
```

Out[183]:
```
(60000, 10000)
```

Helper functions

In [184]:
```
# converts label indices to strings
def get_fashion_mnist_labels(labels):
    """Return text labels for the Fashion-MNIST dataset."""
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]

# outputs
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    """Plot a list of images."""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        ax.imshow(np.array(img))
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes

def evaluate_accuracy(data_iter, net, device=torch.device('cpu')):
    """Evaluate accuracy of a model on the given data set."""
    net.eval()  # Switch to evaluation mode for Dropout, BatchNorm etc l
ayers.
    acc_sum, n = torch.tensor([0], dtype=torch.float32, device=device),
0
    for X, y in data_iter:
        # Copy the data to device.
        X, y = X.to(device), y.to(device)
        with torch.no_grad():
            y = y.long()
            acc_sum += torch.sum((torch.argmax(net(X), dim=1) == y))
            n += y.shape[0]
    return acc_sum.item()/n

X, y = next(iter(data.DataLoader(mnist_train, batch_size=18)))
show_images(X.reshape(18, 28, 28), 2, 9, titles=get_fashion_mnist_labels
(y));
```



Create data iterators

In [185]:
```
batch_size = 256
num_workers = 0

train_iter = data.DataLoader(mnist_train, batch_size, shuffle=True, num_
workers=num_workers)
test_iter = data.DataLoader(mnist_test, batch_size, shuffle=False, num_w
orkers=num_workers)
```

Build the Neural Network

In [186]:
```
net = nn.Sequential(nn.Flatten(),            # input layer: 28*28=784 pi
xels
                    nn.Linear(784, 512),     # 1st hidden layer: 512 out
put features
                    nn.ReLU(),               # 1st hidden layer: ReLU ac
tivation function on each 1st h-layer unit
                    nn.Linear(512, 256),     # 2nd hidden layer: from 51
2 activation units to 256 output features
                    nn.ReLU(),               # 2nd hidden layer: ReLU ac
tivation function on each 2nd h-layer unit
                    nn.Linear(256, 10))      # output layer: from 256 ac
tivation units to 10 digits

def init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights)
```

Out[186]:
```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=512, bias=True)
  (2): ReLU()
  (3): Linear(in_features=512, out_features=256, bias=True)
  (4): ReLU()
  (5): Linear(in_features=256, out_features=10, bias=True)
)
```

2.5 Try 3 different optimization algorithms: SGD, Adam and AdaGrad.

Train and test each optimizer with 3 different stepsizes: 0.1(default), 0.01 and 0.001.

First, define a function for training and testing

In [187]:
```
def train(net, train_iter, test_iter, criterion, num_epochs, batch_size, 
lr, optimizer):
    """Train and evaluate a model with CPU."""
    for epoch in range(num_epochs):
        train_l_sum, train_acc_sum, n = 0.0, 0.0, 0     # set training_lo
ss_sum and train_accuracy_sum to 0
                                                        ## at the start o
f each `epoch`
        for X, y in train_iter:
            optimizer.zero_grad()         # set the gradients to zero bef
ore starting to do backpropragation
                                          ## call `.zero_grad()` again af
ter each `.step()` call

            y_hat = net(X)                # output: PREDICTIONs
            loss = criterion(y_hat, y)    # loss
            loss.backward()               # compute the gradient
            optimizer.step()              # performs a parameter update b
ased on the current gradient
                                          ## (stored in .grad attribute o
f a parameter) and the update rule.

            y = y.type(torch.float32)
            train_l_sum += loss.item()    # LOSS of the training data set
            train_acc_sum += torch.sum((torch.argmax(y_hat, dim=1).type(
torch.FloatTensor) == y).detach()).float()
            n += list(y.size())[0]
        test_acc = evaluate_accuracy(test_iter, net)    # TEST ACCURACY f
or each `epoch`
        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'\
            % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))
```

2.5.1 try SGD optimizer

try the DEFAULT learning rate = 0.1

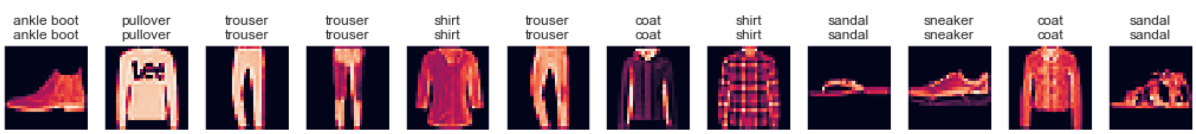
In [188]:

```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.1
# max # of epochs = 10
lr, num_epochs = 0.1, 10

# specify LOSS function
loss = nn.CrossEntropyLoss()

# using SGD optimizer
SGD = torch.optim.SGD(net.parameters(), lr=lr)

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, o
ptimizer=SGD)
```

```
epoch 1, loss 0.0062, train acc 0.411, test acc 0.633
epoch 2, loss 0.0031, train acc 0.707, test acc 0.735
epoch 3, loss 0.0024, train acc 0.777, test acc 0.760
epoch 4, loss 0.0021, train acc 0.806, test acc 0.769
epoch 5, loss 0.0019, train acc 0.826, test acc 0.820
epoch 6, loss 0.0018, train acc 0.834, test acc 0.810
epoch 7, loss 0.0017, train acc 0.846, test acc 0.824
epoch 8, loss 0.0016, train acc 0.853, test acc 0.827
epoch 9, loss 0.0016, train acc 0.857, test acc 0.826
epoch 10, loss 0.0015, train acc 0.863, test acc 0.851
```

In [189]:

```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), ax
is=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(t
rue_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```

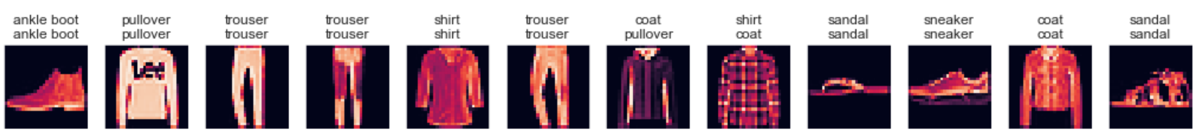


try a SMALLER learning rate = 0.01

In [190]:
```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.01
# max # of epochs = 10
lr, num_epochs = 0.01, 10

# specify LOSS function
loss = nn.CrossEntropyLoss()

# using SGD optimizer
SGD = torch.optim.SGD(net.parameters(), lr=lr)

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, o
ptimizer=SGD)
```

```
epoch 1, loss 0.0092, train acc 0.102, test acc 0.145
epoch 2, loss 0.0087, train acc 0.177, test acc 0.178
epoch 3, loss 0.0082, train acc 0.250, test acc 0.300
epoch 4, loss 0.0064, train acc 0.457, test acc 0.525
epoch 5, loss 0.0050, train acc 0.575, test acc 0.598
epoch 6, loss 0.0043, train acc 0.622, test acc 0.646
epoch 7, loss 0.0038, train acc 0.657, test acc 0.660
epoch 8, loss 0.0035, train acc 0.684, test acc 0.677
epoch 9, loss 0.0033, train acc 0.700, test acc 0.695
epoch 10, loss 0.0031, train acc 0.715, test acc 0.710
```

In [191]:
```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), ax
is=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(t
rue_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```

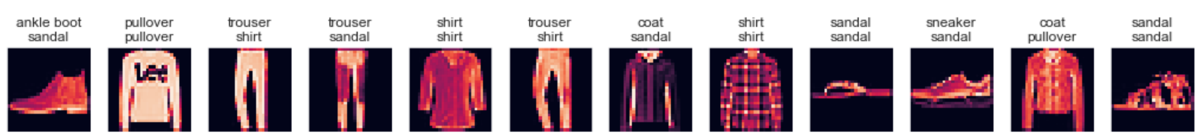


try an EVEN SMALLER learning rate = 0.001

In [192]:
```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.001
# max # of epochs = 10
lr, num_epochs = 0.001, 10

# specify LOSS function
loss = nn.CrossEntropyLoss()

# using SGD optimizer
SGD = torch.optim.SGD(net.parameters(), lr=lr)

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, o
ptimizer=SGD)
```

```
epoch 1, loss 0.0097, train acc 0.100, test acc 0.100
epoch 2, loss 0.0096, train acc 0.100, test acc 0.100
epoch 3, loss 0.0095, train acc 0.100, test acc 0.100
epoch 4, loss 0.0094, train acc 0.100, test acc 0.100
epoch 5, loss 0.0093, train acc 0.100, test acc 0.100
epoch 6, loss 0.0092, train acc 0.100, test acc 0.100
epoch 7, loss 0.0091, train acc 0.100, test acc 0.100
epoch 8, loss 0.0090, train acc 0.102, test acc 0.115
epoch 9, loss 0.0089, train acc 0.143, test acc 0.168
epoch 10, loss 0.0088, train acc 0.177, test acc 0.181
```

In [193]:
```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), ax
is=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(t
rue_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```



- Using SGD optimizer, the default learning rate of 0.1 yields best performances.
- The performance of learning rate of 0.01 is significantly lower than the default learning rate.
- When the learning rate is at 0.001, the test accuracy is improving after every epoch, but very very slowly.
- Therefore, this Neural Network using SGD optimizer is **very sensitive** to the Learning Rate.

2.5.2 try Adam optimizer

try the DEFAULT learning rate = 0.1

In [194]:
```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.1
# max # of epochs = 10
lr, num_epochs = 0.1, 10

# try a DIFFERENT optimizer: Adam
Adam = torch.optim.Adam(net.parameters(), lr=lr)

# specify LOSS function
loss = nn.CrossEntropyLoss()

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, optimizer=Adam)
```

```
epoch 1, loss 0.0342, train acc 0.634, test acc 0.772
epoch 2, loss 0.0025, train acc 0.775, test acc 0.774
epoch 3, loss 0.0022, train acc 0.800, test acc 0.779
epoch 4, loss 0.0023, train acc 0.798, test acc 0.787
epoch 5, loss 0.0022, train acc 0.808, test acc 0.797
epoch 6, loss 0.0022, train acc 0.811, test acc 0.790
epoch 7, loss 0.0021, train acc 0.816, test acc 0.803
epoch 8, loss 0.0020, train acc 0.822, test acc 0.773
epoch 9, loss 0.0021, train acc 0.818, test acc 0.814
epoch 10, loss 0.0021, train acc 0.819, test acc 0.646
```

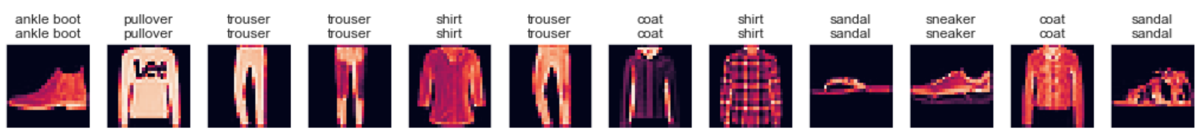
In [195]:
```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), axis=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(true_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```

try a SMALLER learning rate = 0.01

In [196]:
```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.01
# max # of epochs = 10
lr, num_epochs = 0.01, 10

# try a DIFFERENT optimizer: Adam
Adam = torch.optim.Adam(net.parameters(), lr=lr)

# specify LOSS function
loss = nn.CrossEntropyLoss()

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, o
ptimizer=Adam)
```

```
epoch 1, loss 0.0033, train acc 0.703, test acc 0.828
epoch 2, loss 0.0017, train acc 0.848, test acc 0.832
epoch 3, loss 0.0015, train acc 0.861, test acc 0.850
epoch 4, loss 0.0014, train acc 0.869, test acc 0.853
epoch 5, loss 0.0013, train acc 0.875, test acc 0.861
epoch 6, loss 0.0013, train acc 0.876, test acc 0.855
epoch 7, loss 0.0013, train acc 0.880, test acc 0.853
epoch 8, loss 0.0012, train acc 0.884, test acc 0.872
epoch 9, loss 0.0012, train acc 0.888, test acc 0.863
epoch 10, loss 0.0012, train acc 0.890, test acc 0.865
```

In [197]:
```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), ax
is=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(t
rue_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```



try an EVEN SMALLER learning rate = 0.001

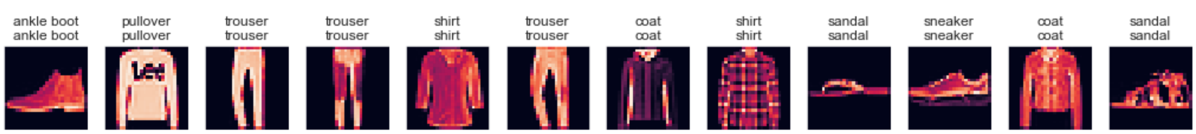
In [198]:

```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.001
# max # of epochs = 10
lr, num_epochs = 0.001, 10

# try a DIFFERENT optimizer: Adam
Adam = torch.optim.Adam(net.parameters(), lr=lr)

# specify LOSS function
loss = nn.CrossEntropyLoss()

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, o
ptimizer=Adam)
```

```
epoch 1, loss 0.0062, train acc 0.477, test acc 0.725
epoch 2, loss 0.0025, train acc 0.764, test acc 0.788
epoch 3, loss 0.0022, train acc 0.803, test acc 0.803
epoch 4, loss 0.0020, train acc 0.819, test acc 0.813
epoch 5, loss 0.0019, train acc 0.833, test acc 0.826
epoch 6, loss 0.0018, train acc 0.843, test acc 0.835
epoch 7, loss 0.0017, train acc 0.849, test acc 0.836
epoch 8, loss 0.0016, train acc 0.853, test acc 0.841
epoch 9, loss 0.0016, train acc 0.857, test acc 0.842
epoch 10, loss 0.0015, train acc 0.860, test acc 0.844
```

In [199]:

```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), ax
is=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(t
rue_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```



- Using Adam optimizer, the learning rate of 0.01 yields the best performance.
- However, the training accuracies and testing accuracies are **NOT sensitive** to different learning rates.

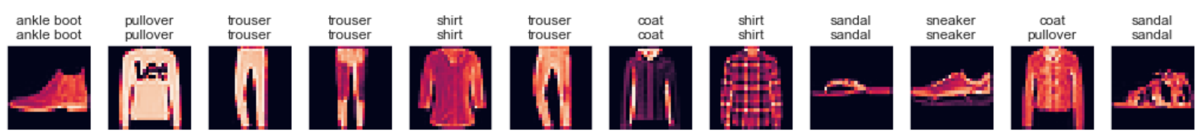
2.5.3 try AdaGrad optimizer

try the DEFAULT learning rate = 0.1

In [200]:
```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.1
# max # of epochs = 10
lr, num_epochs = 0.1, 10

# try a DIFFERENT optimizer: Adam
AdaGrad = torch.optim.Adagrad(net.parameters(), lr=lr)

# specify LOSS function
loss = nn.CrossEntropyLoss()

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, o
ptimizer=AdaGrad)
```

```
epoch 1, loss 0.0055, train acc 0.488, test acc 0.559
epoch 2, loss 0.0029, train acc 0.710, test acc 0.680
epoch 3, loss 0.0026, train acc 0.743, test acc 0.739
epoch 4, loss 0.0023, train acc 0.780, test acc 0.763
epoch 5, loss 0.0021, train acc 0.806, test acc 0.786
epoch 6, loss 0.0020, train acc 0.819, test acc 0.795
epoch 7, loss 0.0019, train acc 0.826, test acc 0.805
epoch 8, loss 0.0019, train acc 0.830, test acc 0.819
epoch 9, loss 0.0018, train acc 0.835, test acc 0.796
epoch 10, loss 0.0018, train acc 0.839, test acc 0.813
```

In [201]:
```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), ax
is=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(t
rue_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```

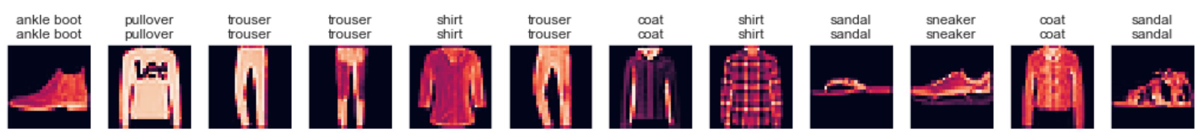


try a SMALLER learning rate = 0.01

In [202]:
```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.01
# max # of epochs = 10
lr, num_epochs = 0.01, 10

# try a DIFFERENT optimizer: Adam
AdaGrad = torch.optim.Adagrad(net.parameters(), lr=lr)

# specify LOSS function
loss = nn.CrossEntropyLoss()

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, o
ptimizer=AdaGrad)
```

```
epoch 1, loss 0.0037, train acc 0.659, test acc 0.764
epoch 2, loss 0.0023, train acc 0.788, test acc 0.787
epoch 3, loss 0.0021, train acc 0.810, test acc 0.798
epoch 4, loss 0.0020, train acc 0.823, test acc 0.805
epoch 5, loss 0.0018, train acc 0.833, test acc 0.825
epoch 6, loss 0.0018, train acc 0.839, test acc 0.831
epoch 7, loss 0.0017, train acc 0.843, test acc 0.818
epoch 8, loss 0.0017, train acc 0.848, test acc 0.829
epoch 9, loss 0.0016, train acc 0.852, test acc 0.835
epoch 10, loss 0.0016, train acc 0.855, test acc 0.838
```

In [203]:
```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), ax
is=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(t
rue_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```

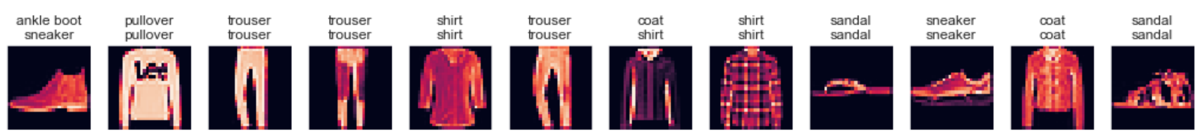


try an EVEN SMALLER learning rate = 0.001

In [204]:
```
# reinitializing WEIGHTS
net.apply(init_weights)

# learning rate = 0.001
# max # of epochs = 10
lr, num_epochs = 0.001, 10

# try a DIFFERENT optimizer: Adam
AdaGrad = torch.optim.Adagrad(net.parameters(), lr=lr)

# specify LOSS function
loss = nn.CrossEntropyLoss()

train(net, train_iter, test_iter, loss, num_epochs, batch_size, lr=lr, optimizer=AdaGrad)
```

```
epoch 1, loss 0.0101, train acc 0.165, test acc 0.210
epoch 2, loss 0.0075, train acc 0.271, test acc 0.344
epoch 3, loss 0.0064, train acc 0.399, test acc 0.496
epoch 4, loss 0.0054, train acc 0.539, test acc 0.562
epoch 5, loss 0.0047, train acc 0.592, test acc 0.603
epoch 6, loss 0.0042, train acc 0.626, test acc 0.631
epoch 7, loss 0.0039, train acc 0.649, test acc 0.648
epoch 8, loss 0.0037, train acc 0.666, test acc 0.667
epoch 9, loss 0.0036, train acc 0.679, test acc 0.677
epoch 10, loss 0.0034, train acc 0.688, test acc 0.684
```

In [205]:
```
X, y = next(iter(test_iter))
true_labels = get_fashion_mnist_labels(y)
pred_labels = get_fashion_mnist_labels(np.argmax(net(X).data.numpy(), axis=1))
titles = [truelabel + '\n' + predlabel for truelabel, predlabel in zip(true_labels, pred_labels)]
show_images(X.reshape(256, 28, 28), 1, 12, titles=titles);
```



- Using AdaGrad optimizer, the learning rate of 0.01 yields the best performance.
- When using a very small learning rate of 0.001, the training and testing accuracies are very low.
- Therefore, AdaGrad optimizer is **very sensitive** to the Learning Rate.

2.6 Pick the best setup based on the highest training accuracy above, and check if it has the highest testing accuracy.

Pick the training and testing accuracies at the 10th epoch for each algorithm with a different learning rate.

In [225]:
```
df=pd.DataFrame(np.array([['SGD', '0.1', 0.863, 0.851],
                          ['SGD', '0.01', 0.715, 0.710],
                          ['SGD', '0.001', 0.177, 0.181],
                          ['Adam', '0.1', 0.819, 0.646],
                          ['Adam', '0.01', 0.890, 0.865],
                          ['Adam', '0.001', 0.860, 0.844],
                          ['AdaGrad', '0.1', 0.839, 0.813],
                          ['AdaGrad', '0.01', 0.855, 0.838],
                          ['AdaGrad', '0.001', 0.688, 0.684]]),
                columns=['Optimizer','Learning Rate','Training Accuracy',
'Testing Accuracy'])
df=df.astype({'Training Accuracy': 'float64','Testing Accuracy': 'float6
4'})
df
```

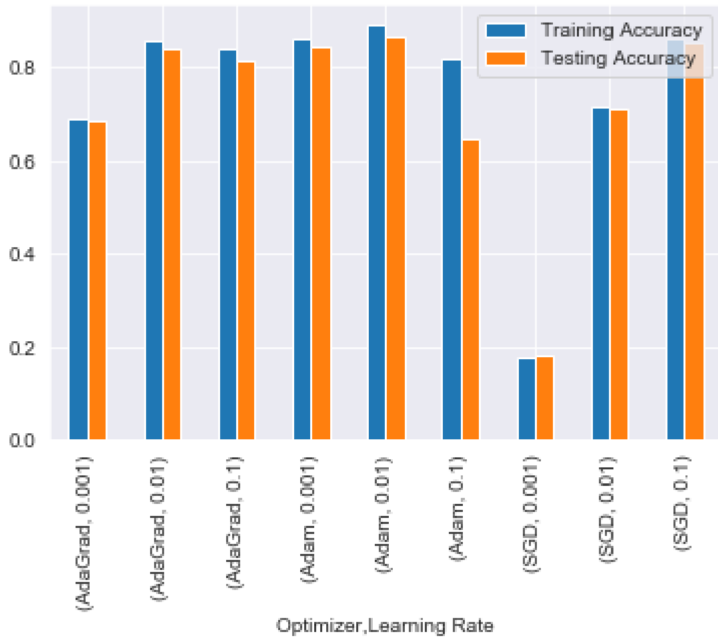
Out[225]:

	Optimizer	Learning Rate	Training Accuracy	Testing Accuracy
0	SGD	0.1	0.863	0.851
1	SGD	0.01	0.715	0.710
2	SGD	0.001	0.177	0.181
3	Adam	0.1	0.819	0.646
4	Adam	0.01	0.890	0.865
5	Adam	0.001	0.860	0.844
6	AdaGrad	0.1	0.839	0.813
7	AdaGrad	0.01	0.855	0.838
8	AdaGrad	0.001	0.688	0.684

Plot a bar chart to compare the training and testing accuracies from different algorithms with different learning rates.

In [226]:
```
df.groupby(['Optimizer','Learning Rate']).sum().plot(kind='bar')
```

Out[226]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fa94a3946d0>
```



Check if the setup with highest `Training Accuracy` has the highest `Testing Accuracy`.

In [233]:
```
print(df.iloc[df['Training Accuracy'].idxmax()])
print(df.iloc[df['Testing Accuracy'].idxmax()])
```

```
Optimizer              Adam
Learning Rate          0.01
Training Accuracy      0.89
Testing Accuracy      0.865
Name: 4, dtype: object
Optimizer              Adam
Learning Rate          0.01
Training Accuracy      0.89
Testing Accuracy      0.865
Name: 4, dtype: object
```

Yes, using **Adam optimizer with learning rate of 0.01** that is slightly smaller than the default learning rate 0.1 led to both the highest training accuracy and testing accuracy.

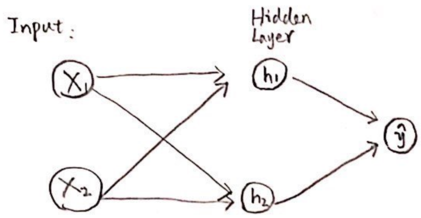
Question 3 Function Approximation

3.1 Prove that the ReLU network is a piecewise-linear function.

In [235]:
```
Q_31 = Image(filename=('Q31.jpg'))
Q_31
```

Out[235]:

A ReLU network that has a single hidden layer with 2 neurons.



$$\begin{cases} h = \sigma(W^{(1)}x + b^{(1)}) \\ \hat{y} = W^{(2)}h + b^{(2)} \end{cases}$$

$x \in \mathbb{R}^2$, $W^{(1)} \in \mathbb{R}^{2\times 2}$, $b^{(1)} \in \mathbb{R}^{(2)}$

$y \in \mathbb{R}$, $W^{(2)} \in \mathbb{R}^{1\times 2}$, $b^{(2)} \in \mathbb{R}$

σ is the ReLU activation function,

$\sigma(z) = \max(z, 0)$, z is a vector.

Let: $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$,

$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \end{bmatrix}$, $b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$, $W^{(2)} = [W_{11}^{(2)} \quad W_{12}^{(2)}]$

$$h = \sigma\left(\begin{bmatrix} W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)} \\ W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)} \end{bmatrix}\right) = \begin{bmatrix} \max(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)}, 0) \\ \max(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)}, 0) \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

$$\boxed{\hat{y} = W_{11}^{(2)} \cdot \max(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)}, 0) + W_{12}^{(2)} \cdot \max(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)}, 0) + b^{(2)}}$$

Since $h_1 = \max(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)}, 0)$ and $h_2 = \max(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)}, 0)$ are both ReLU functions that are Piecewise-Linear,

$\hat{y}$ as a linear combination of 2 Piecewise-Linear functions is also Piecewise-Linear.

To specify the 4 set of pieces and the value of $\hat{y}$ on each piece:

$$\hat{y} = \begin{cases} W_{12}^{(2)} \cdot (W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)}) + b^{(2)} & \text{if } W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)} < 0 \text{ and } W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)} > 0 \\ b^{(2)} & \text{if } W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)} < 0 \text{ and } W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)} < 0 \\ W_{11}^{(2)} \cdot (W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)}) + b^{(2)} & \text{if } W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)} > 0 \text{ and } W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)} < 0 \\ W_{11}^{(2)} \cdot (W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)}) + W_{12}^{(2)} \cdot (W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)}) + b^{(2)} & \text{if } W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + b_1^{(1)} > 0 \text{ and } W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + b_2^{(1)} > 0 \end{cases}$$

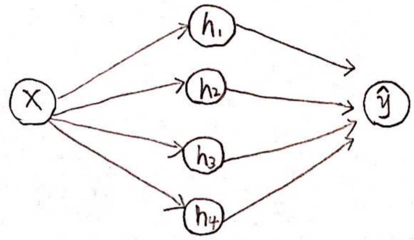
3.2 Represent the continuous piecewise-linear function with a ReLU Neural Network that uses a single hidden layer.

In [236]:

```
Q_32 = Image(filename=('Q32.jpg'))
Q_32
```

Out[236]:

A ReLU network that uses a single hidden layer with 4 neurons.



$$\begin{cases} h = \sigma(W^{(1)}x + b^{(1)}) \\ \hat{y} = W^{(2)}h + b^{(2)} \end{cases}$$

σ is the ReLU activation function : $\sigma(z) = \max(z, 0)$, z is a vector.

$x \in \mathbb{R}$, $W^{(1)} \in \mathbb{R}^4$, $b^{(1)} \in \mathbb{R}^4$, $y \in \mathbb{R}$, $W^{(2)} \in \mathbb{R}^{1\times 4}$, $b^{(2)} \in \mathbb{R}$

The parameters are the following:

$$W^{(1)} = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \; b^{(1)} = \begin{bmatrix} -3 \\ 3 \\ -5 \\ -10 \end{bmatrix}, \quad W^{(2)} = [-1 \;\; 1 \;\; 1 \;\; -3], \; b^{(2)} = 0$$

The resulting piece-wise linear function is:

$$\hat{y} = [-1 \;\; 1 \;\; 1 \;\; -3] \;\sigma\left(\begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} x + \begin{bmatrix} -3 \\ 3 \\ -5 \\ -10 \end{bmatrix}\right) + 0$$

$$= [-1 \;1 \;1 \;-3] \begin{bmatrix} \sigma(-x-3) \\ \sigma(x+3) \\ \sigma(x-5) \\ \sigma(x-10) \end{bmatrix} + 0$$

$$= -1\cdot\max(-x-3, 0) + 1\cdot\max(x+3, 0) + 1\cdot\max(x-5, 0) - 3\cdot\max(x-10, 0) + 0$$

$\Updownarrow$ Equivalent to

$$f(x) = \begin{cases} x+3 & \text{if } x < 5 \\ 2x-2 & \text{if } 5 \le x < 10 \\ -1x+28 & \text{if } 10 \le x \end{cases}$$