kjp:

Derived from https://www.tensorflow.org/tutorials/generative/autoencoder (https://www.tensorflow.org/tutorials/generative/autoencoder)

In [71]:
```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Intro to Autoencoders

[View on TensorFlow.org (https://www.tensorflow.org/tutorials/generative/autoencoder)](https://www.tensorflow.org/tutorials/generative/autoencoder) [Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials)

This tutorial introduces autoencoders with three examples: the basics, image denoising, and anomaly detection.

An autoencoder is a special type of neural network that is trained to copy its input to its output. For example, given an image of a handwritten digit, an autoencoder first encodes the image into a lower dimensional latent representation, then decodes the latent representation back to an image. An autoencoder learns to compress the data while minimizing the reconstruction error.

To learn more about autoencoders, please consider reading chapter 14 from [Deep Learning (https://www.deeplearningbook.org/)](https://www.deeplearningbook.org/) by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

Import TensorFlow and other libraries

In [72]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf

from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, losses
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Model
```

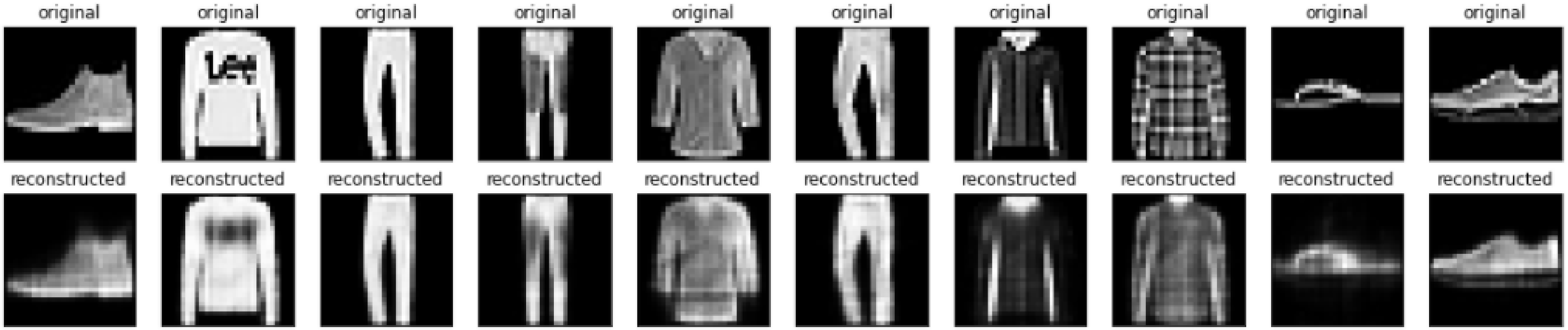
Load the dataset

To start, you will train the basic autoencoder using the Fashion MNIST dataset. Each image in this dataset is 28x28 pixels.

In [73]:
```
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

print (x_train.shape)
print (x_test.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
```

First example: Basic autoencoder



Define an autoencoder with two Dense layers: an `encoder`, which compresses the images into a 64 dimensional latent vector, and a `decoder`, that reconstructs the original image from the latent space.

To define your model, use the [Keras Model Subclassing API (https://www.tensorflow.org/guide/keras/custom_layers_and_models)](https://www.tensorflow.org/guide/keras/custom_layers_and_models).

In [74]:
```
latent_dim = 64

class Autoencoder(Model):
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([
      layers.Flatten(),
      layers.Dense(latent_dim, activation='relu'),
    ])
    self.decoder = tf.keras.Sequential([
      layers.Dense(784, activation='sigmoid'),
      layers.Reshape((28, 28))
    ])

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

autoencoder = Autoencoder(latent_dim)
```

In [75]:
```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

Train the model using `x_train` as both the input and the target. The `encoder` will learn to compress the dataset from 784 dimensions to the latent space, and the `decoder` will learn to reconstruct the original images. .

In [76]:
```
autoencoder.fit(x_train, x_train,
                epochs=10,
                shuffle=True,
                validation_data=(x_test, x_test))
```

```
Epoch 1/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.0238 - val_
loss: 0.0134
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0118 - val_
loss: 0.0108
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0102 - val_
loss: 0.0099
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0096 - val_
loss: 0.0095
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0093 - val_
loss: 0.0095
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0092 - val_
loss: 0.0092
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0091 - val_
loss: 0.0092
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0090 - val_
loss: 0.0091
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0090 - val_
loss: 0.0093
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0089 - val_
loss: 0.0089
```

Out[76]: <keras.callbacks.History at 0x7f98b9273750>

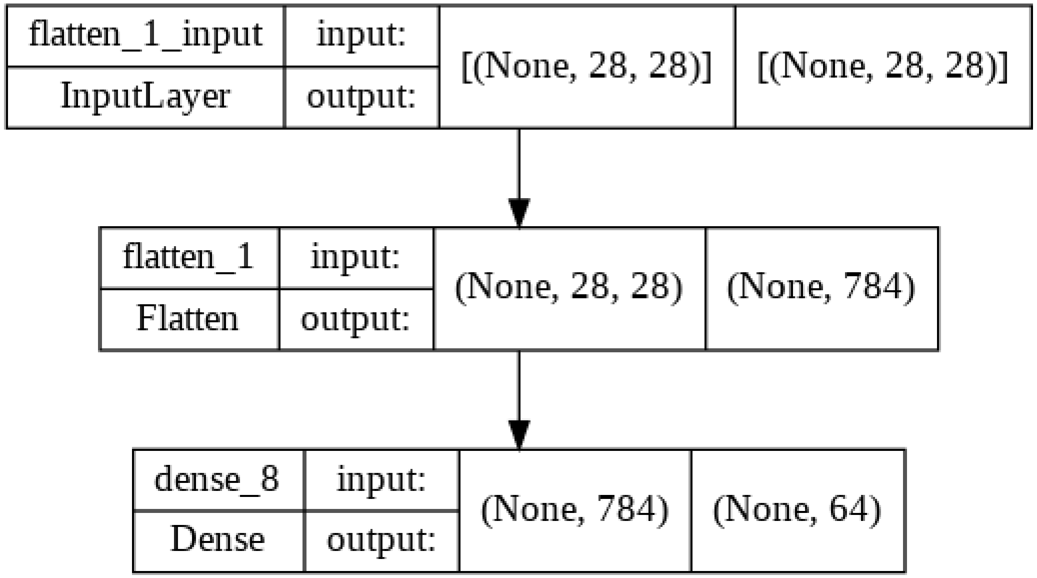
Now that the model is trained, let's test it by encoding and decoding images from the test set.

In [77]:
```
from tensorflow.keras.utils import plot_model

import os
import tempfile

tempdir = tempfile.gettempdir()
```

In [78]:

```
plot_model(autoencoder.encoder, to_file=os.path.join(tempdir, "autoencoder_simple_encoder.png"), show_shapes=True)
```

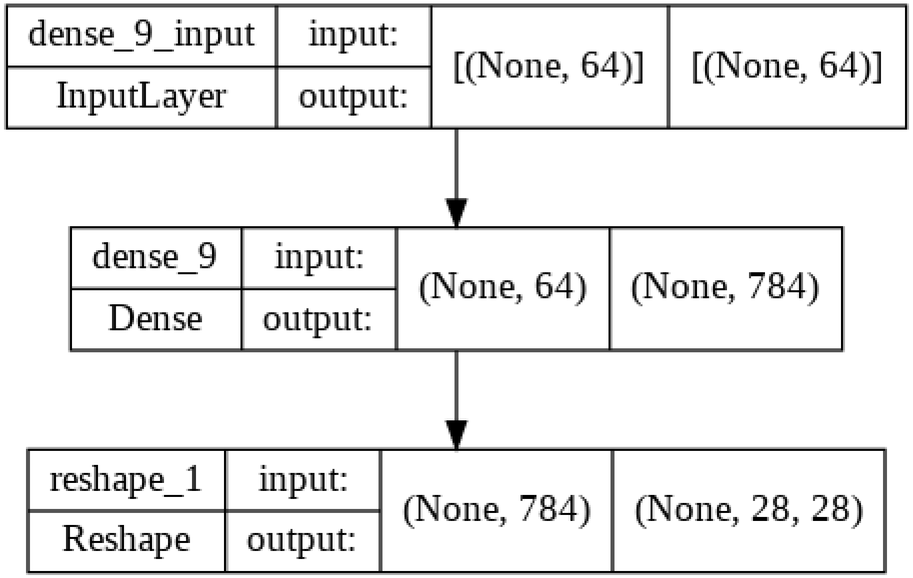
Out[78]:



In [79]:

```
plot_model(autoencoder.decoder, to_file=os.path.join(tempdir, "autoencoder_simpl
e_decoder.png"), show_shapes=True)
```

Out[79]:



In [80]:
```
ae_encoder_dir = tempfile.mkdtemp()
autoencoder.encoder.save(ae_encoder_dir)

ae_decoder_dir = tempfile.mkdtemp()
autoencoder.decoder.save(ae_decoder_dir)
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have ye
t to be built. `model.compile_metrics` will be empty until you train or evalua
te the model.
INFO:tensorflow:Assets written to: /tmp/tmpk_njncjr/assets
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have ye
t to be built. `model.compile_metrics` will be empty until you train or evalua
te the model.
INFO:tensorflow:Assets written to: /tmp/tmp_33waxl6/assets
```

Save the model

In [81]:
```
encoder_rest = tf.keras.models.load_model(ae_encoder_dir)
decoder_rest = tf.keras.models.load_model(ae_decoder_dir)
```

```
WARNING:tensorflow:No training configuration found in save file, so the model
was *not* compiled. Compile it manually.
WARNING:tensorflow:No training configuration found in save file, so the model
was *not* compiled. Compile it manually.
```

In [81]:
```
```

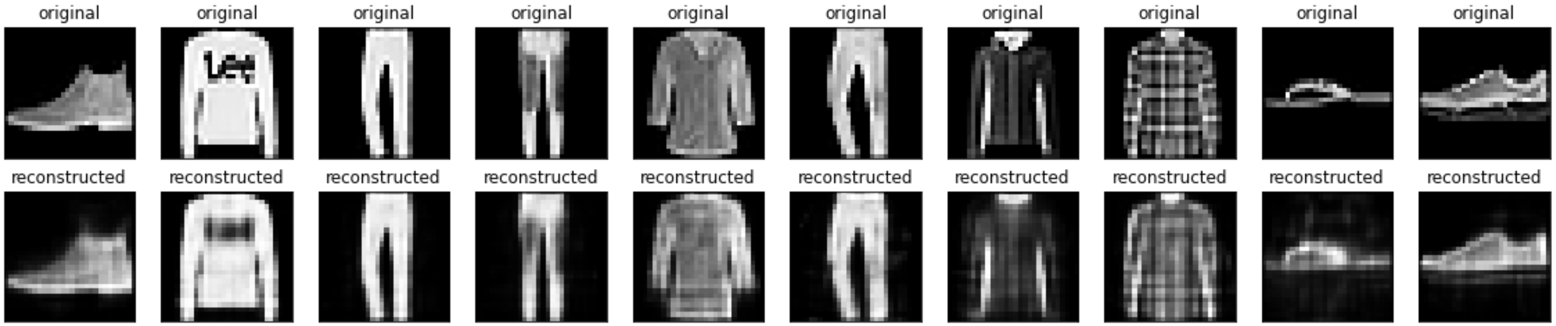
In [82]:
```
encoded_imgs = autoencoder.encoder(x_test).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()
```

In [83]:

```
encoded_imgs = encoder_rest(x_test).numpy()
decoded_imgs = decoder_rest(encoded_imgs).numpy()
```

In [84]:

```
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
  # display original
  ax = plt.subplot(2, n, i + 1)
  plt.imshow(x_test[i])
  plt.title("original")
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)

  # display reconstruction
  ax = plt.subplot(2, n, i + 1 + n)
  plt.imshow(decoded_imgs[i])
  plt.title("reconstructed")
  plt.gray()
  ax.get_xaxis().set_visible(False)
  ax.get_yaxis().set_visible(False)
plt.show()
```



Examine the latent representations of the test dataset

In [85]:

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import matplotlib as mpl

def PCA_fit(X, n_components=2):
  pca = PCA(n_components=n_components)
  pca.fit(X)

  return pca

default_cmap = "plasma"

def plot_2D(X, y, fig=None, ax=None, title=None, visible=True, save_file=None,
cmap_name=default_cmap, colorbar=True, alpha=0.9):
  if fig==None and ax==None:
    fig, ax = plt.subplots( )

  cmap=plt.cm.get_cmap(cmap_name, np.unique(y).shape[0])

  ax_res = ax.scatter(X[:, 0], X[:, 1],
                      c=y, edgecolor='none', alpha=alpha,
                      cmap=cmap)
  ax.set_xlabel('component 1')
  ax.set_ylabel('component 2')

  if colorbar:
    cmap=plt.cm.get_cmap(default_cmap, np.unique(y).shape[0])

    norm = mpl.colors.Normalize(vmin=0,vmax=9)

    plt.colorbar( plt.cm.ScalarMappable(norm=norm, cmap=cmap), ax=ax)

  if title is not None:
    ax.set_title(title)
```

```
  if save_file is not None:
    fig.savefig(save_file)

  if not visible:
    plt.close(fig)




  return fig, ax

def plot_cond_2d(X, y, cmap_name=default_cmap):
  y_unique= np.unique(y)
  x0_min, x0_max = X[:,0].min(), X[:, 0].max()
  x1_min, x1_max = X[:, ].min(), X[:, 1].max()

  num_per_row = 5
  num_rows = int( y_unique.shape[0]/num_per_row + 0.5)

  fig, axs = plt.subplots(num_rows, num_per_row, figsize=(20,12))
  axs = axs.ravel()

  for i, y_val in enumerate(y_unique):
    ax = axs[i]
    ax.set_xlim(x0_min, x0_max)
    ax.set_ylim(x1_min, x1_max)

    X_proj_val = X[ y == y_val, :]
    y_proj_val = y[ y == y_val ]

    plot_2D(X_proj_val, y_proj_val, fig=fig, ax=ax, title=f"y = {y_val}", colorb
ar=False)

  fig.tight_layout()

  return fig, axs
```

In [86]:

```
encoded_imgs.shape
```

Out[86]:

```
(10000, 64)
```

Project the high dimensionality latents into 2D

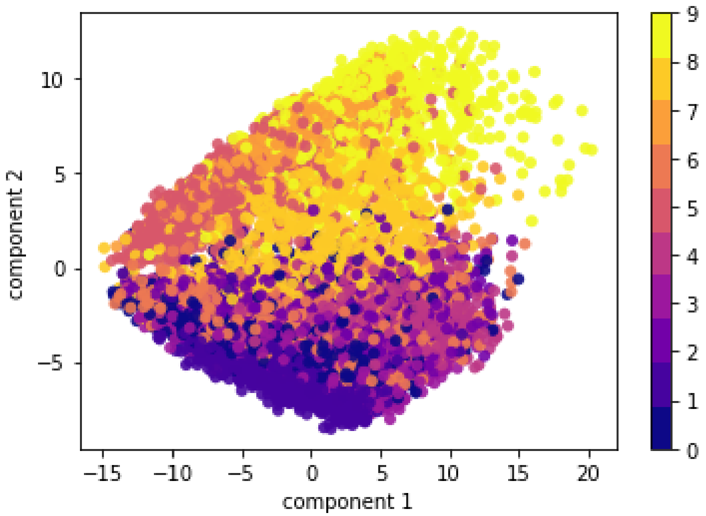
In [87]:

```
pca = PCA_fit(encoded_imgs, n_components=10)
X_proj = pca.transform(encoded_imgs)

X_proj.shape
```

Out[87]:

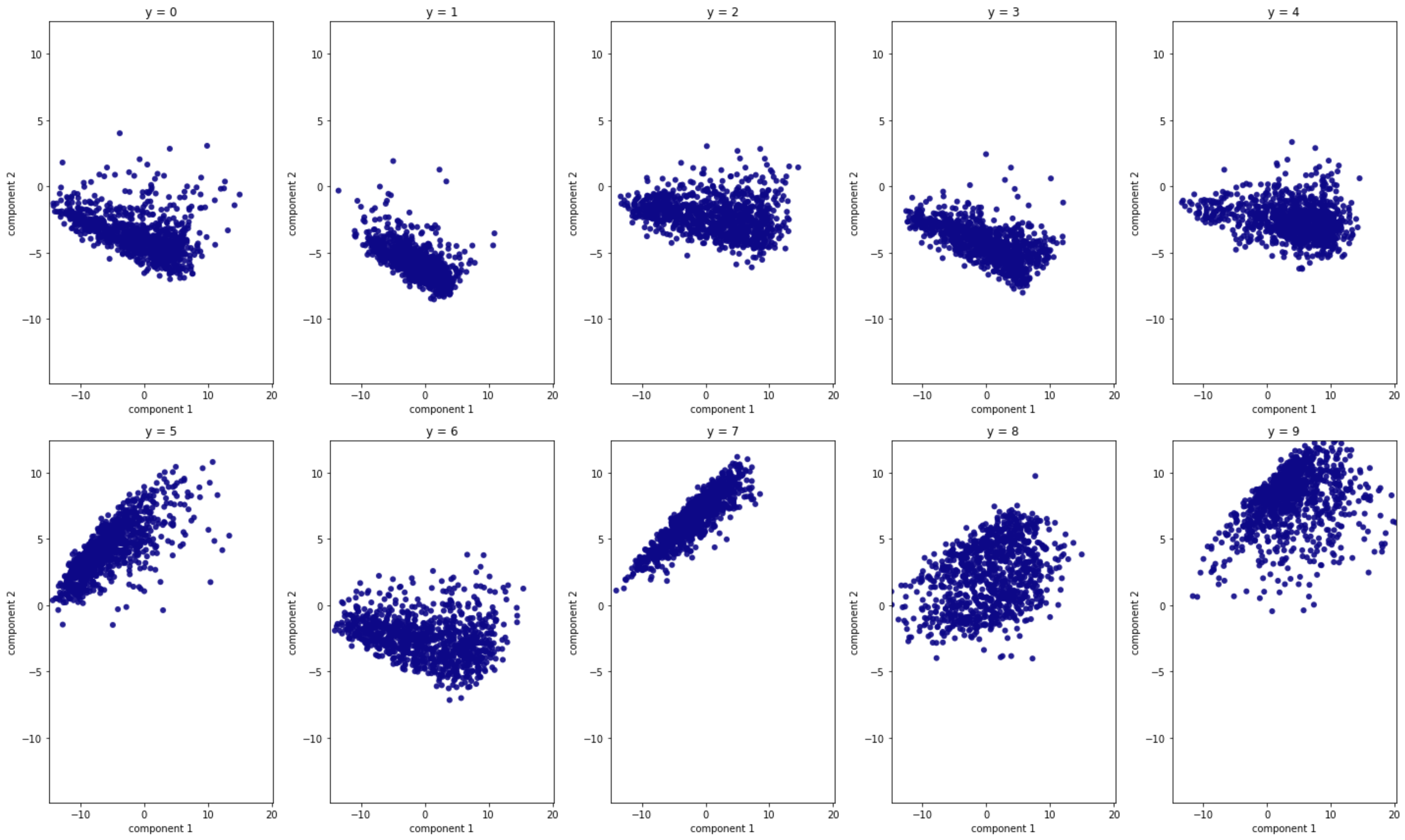
```
(10000, 10)
```

In [88]:
```
y_proj = y_test

fig, ax = plot_2D(X_proj, y_test)

fig.savefig( os.path.join(tempdir, "autoencoder_latents.png"))
```



Zoom in on the latents: separate plots per class

In [89]:
```
fig, axs = plot_cond_2d(X_proj, y_proj)
```

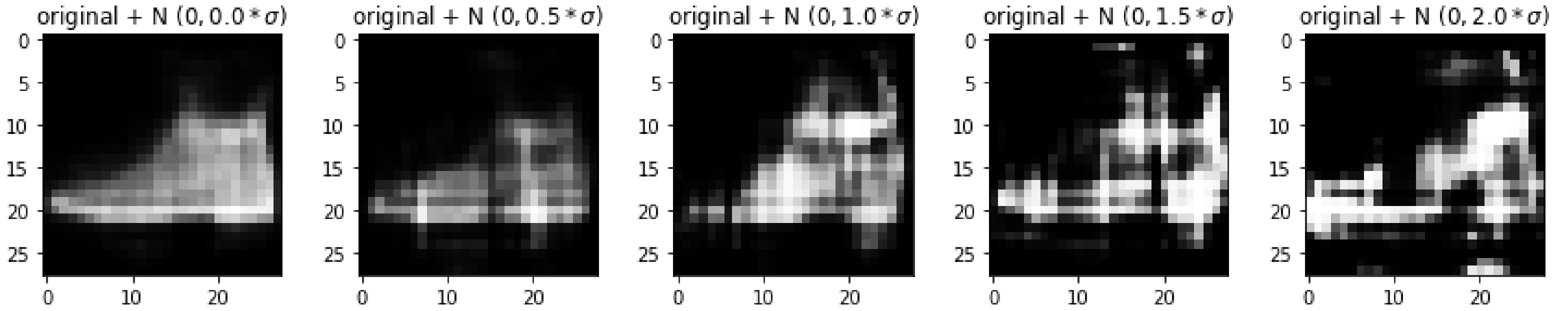


In [90]:
```
fig.savefig( os.path.join(tempdir, "autoencoder_latents_by_target.png"))
```

Explore the latents in a small radius of the latent of a single input

In [91]:

```
img_start= x_test[0,: ]
encoded_img_start = autoencoder.encoder( np.expand_dims(img_start, axis=0)).nump
y()

encoded_imgs_mean, encoded_imgs_std = np.mean(encoded_imgs, axis=0), np.std(enco
ded_imgs, axis=0)



range_max, steps = 2, 5

fig, axs = plt.subplots(1, steps, figsize=(12,10))

for i, frac in enumerate(np.linspace(0, range_max, steps)):
  encoded_img_end = encoded_img_start + np.random.normal( loc=0.0, scale=frac *
encoded_imgs_std )
  img_end = autoencoder.decoder(encoded_img_end).numpy()[0]

  ax = axs[i]

  _= ax.imshow(img_end)
  _= ax.set_title(f"original + N $(0, {frac} * \sigma)$")

fig.tight_layout()

fig.savefig( os.path.join(tempdir, "autoencoder_perturb_single_img.png"))
```

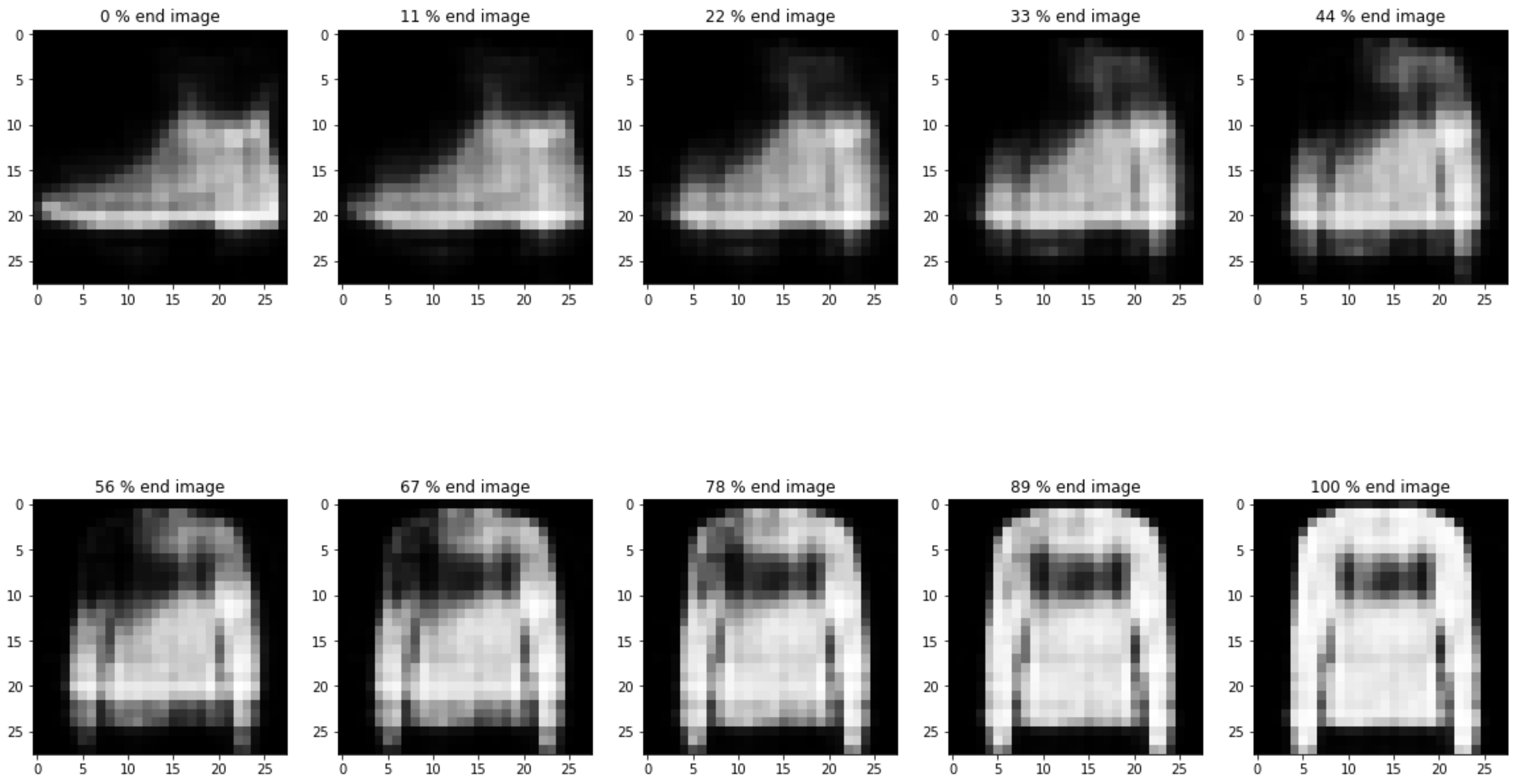


Interpolate between the latents of two inputs

In [92]:

```
def interpolate_imgs(img_start, img_end, steps=10):

  encoded_img_start = autoencoder.encoder( np.expand_dims(img_start, axis=0)).nu
mpy()
  encoded_img_end =   autoencoder.encoder( np.expand_dims(img_end,   axis=0)).nu
mpy()

  encoded_imgs_interp = [ (1 - w/(steps-1))*encoded_img_start + (w/(steps-1))*e
ncoded_img_end for w in range(0,steps-1) ]
  encoded_imgs_interp.append(encoded_img_end)

  num_per_row = 5
  num_rows = int( len(encoded_imgs_interp)/num_per_row + 0.5)

  fig, axs = plt.subplots(num_rows, num_per_row, figsize=(20,12))
  axs = axs.ravel()

  for i, encoded_img in enumerate(encoded_imgs_interp):
    ax = axs[i]
    img = autoencoder.decoder(encoded_img).numpy()[0]
    ax.imshow(img)

    ax.set_title(f"{round(100 * i/(steps-1))} % end image")

  return fig, axs
```

In [93]:
```
fig, axs = interpolate_imgs( x_test[0], x_test[1])
fig.savefig( os.path.join(tempdir, "autoencoder_interpolate_2_imgs.png"))
```

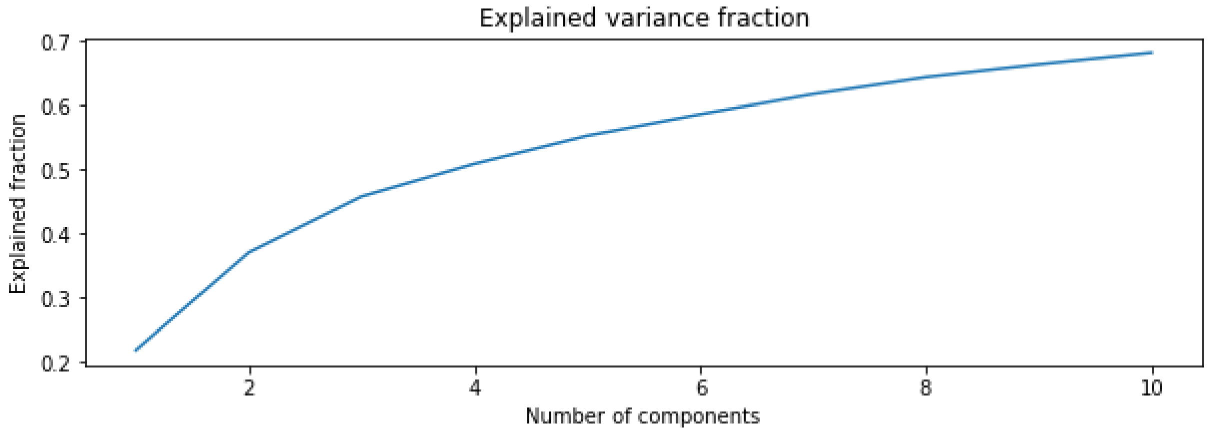


Examine the 2D projections obtained by PCA on the high dimensionality latents

In [94]:
```
pca_comp = pca.components_
pca_comp.shape
```

Out[94]:
```
(10, 64)
```

In [95]:
```
fig, ax = plt.subplots(1,1, figsize=(10,3))
_= ax.plot( range(1, pca_comp.shape[0] +1), np.cumsum(pca.explained_variance_ra
tio_) )
_= ax.set_title("Explained variance fraction")
_= ax.set_xlabel("Number of components")
_= ax.set_ylabel("Explained fraction")
```



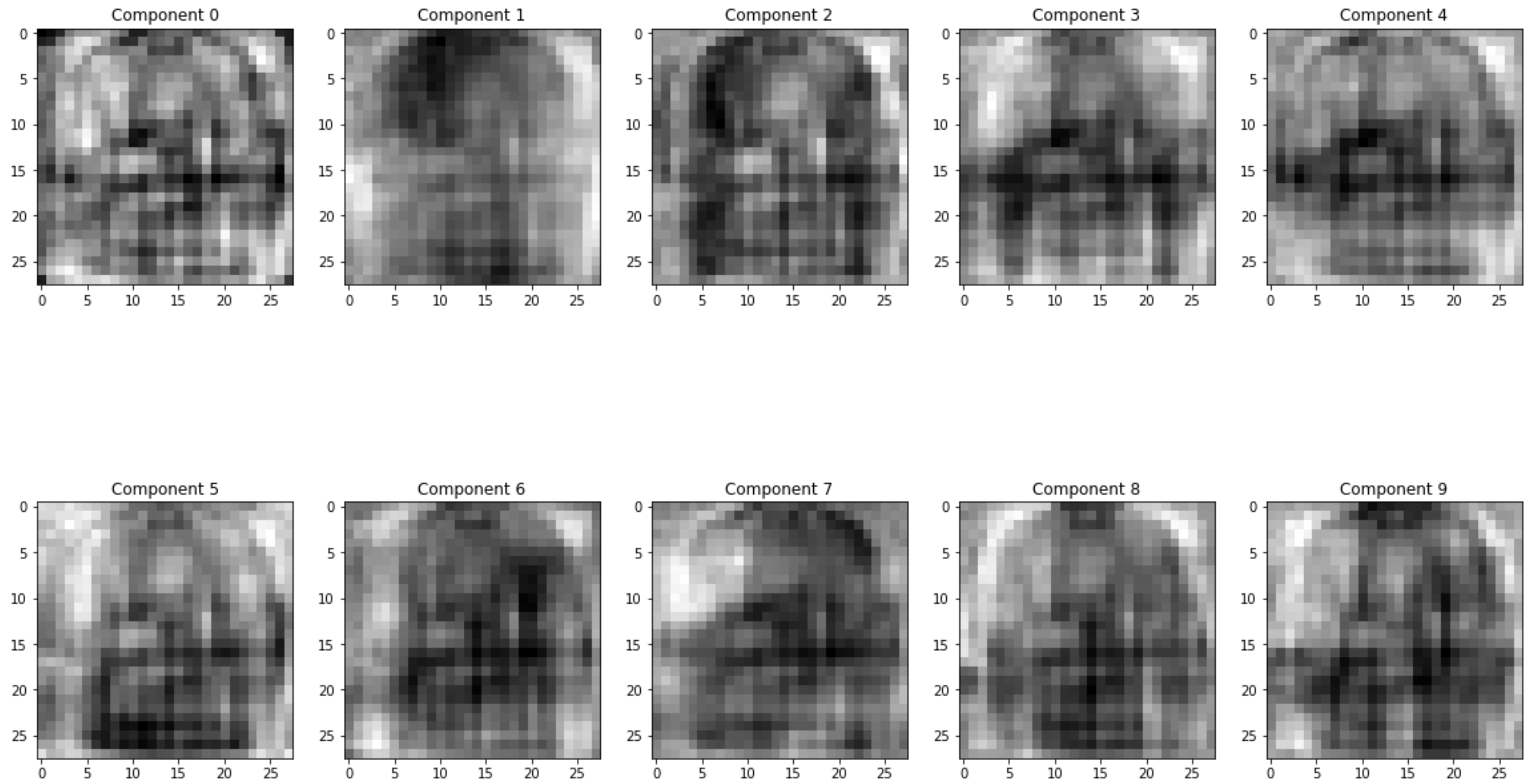
In [96]:
```
pca_imgs = autoencoder.decoder(pca_comp).numpy()
```

In [97]:
```
pca_imgs.shape
```

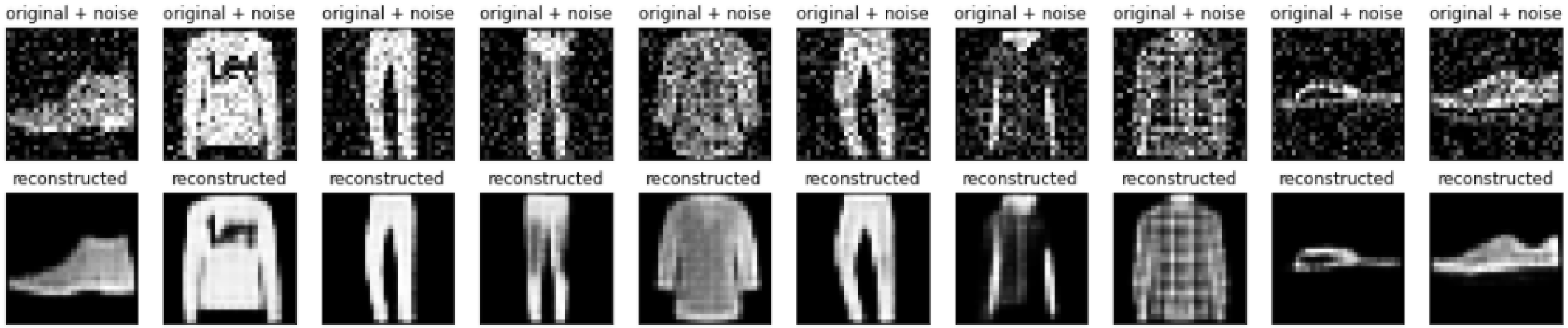
Out[97]:
```
(10, 28, 28)
```

In [98]:

```
num_per_row = 5
num_rows = int( pca_imgs.shape[0]/num_per_row + 0.5)

fig, axs = plt.subplots(num_rows, num_per_row, figsize=(20,12))
axs = axs.ravel()

for i, pca_img in enumerate(pca_imgs):
  ax = axs[i]
  _= ax.imshow(pca_img)

  ax.set_title(f"Component {i}")

fig.savefig( os.path.join(tempdir, "autoencoder_latents_components.png"))
```



Second example: Image denoising



An autoencoder can also be trained to remove noise from images. In the following section, you will create a noisy version of the Fashion MNIST dataset by applying random noise to each image. You will then train an autoencoder using the noisy image as input, and the original image as the target.

Let's reimport the dataset to omit the modifications made earlier.

In [99]:
```
(x_train, _), (x_test, _) = fashion_mnist.load_data()
```

In [100]:
```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]

print(x_train.shape)
```

```
(60000, 28, 28, 1)
```

Adding random noise to the images

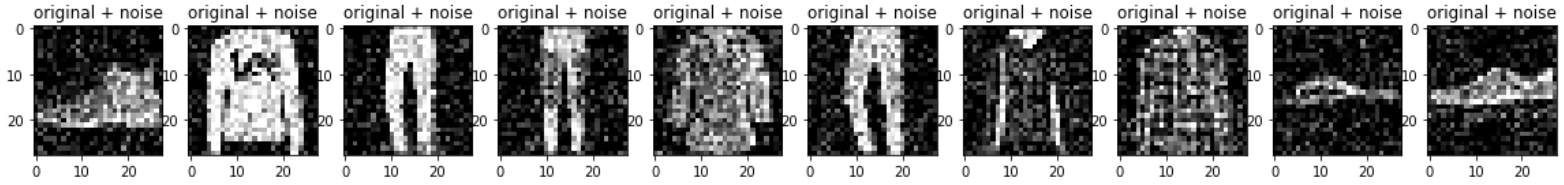
In [101]:

```
noise_factor = 0.2
x_train_noisy = x_train + noise_factor * tf.random.normal(shape=x_train.shape)
x_test_noisy = x_test + noise_factor * tf.random.normal(shape=x_test.shape)

x_train_noisy = tf.clip_by_value(x_train_noisy, clip_value_min=0., clip_value_ma
x=1.)
x_test_noisy = tf.clip_by_value(x_test_noisy, clip_value_min=0., clip_value_max=
1.)
```

Plot the noisy images.

In [102]:

```
n = 10
plt.figure(figsize=(20, 2))
for i in range(n):
    ax = plt.subplot(1, n, i + 1)
    plt.title("original + noise")
    plt.imshow(tf.squeeze(x_test_noisy[i]))
    plt.gray()
plt.show()
```



Define a convolutional autoencoder

In this example, you will train a convolutional autoencoder using Conv2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) layers in the `encoder`, and Conv2DTranspose (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose) layers in the `decoder`.

In [103]:

```
class Denoise(Model):
  def __init__(self):
    super(Denoise, self).__init__()
    self.encoder = tf.keras.Sequential([
      layers.Input(shape=(28, 28, 1)),
      layers.Conv2D(16, (3, 3), activation='relu', padding='same', strides=2),
      layers.Conv2D(8, (3, 3), activation='relu', padding='same', strides=2)])

    self.decoder = tf.keras.Sequential([
      layers.Conv2DTranspose(8, kernel_size=3, strides=2, activation='relu', pad
ding='same'),
      layers.Conv2DTranspose(16, kernel_size=3, strides=2, activation='relu', pa
dding='same'),
      layers.Conv2D(1, kernel_size=(3, 3), activation='sigmoid', padding='sam
e')])

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

autoencoder = Denoise()
```

In [104]:

```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

In [105]:

```
autoencoder.fit(x_train_noisy, x_train,
                epochs=10,
                shuffle=True,
                validation_data=(x_test_noisy, x_test))
```

```
Epoch 1/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0186 - val_
loss: 0.0110
Epoch 2/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0101 - val_
loss: 0.0093
Epoch 3/10
1875/1875 [==============================] - 7s 3ms/step - loss: 0.0091 - val_
loss: 0.0089
Epoch 4/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0087 - val_
loss: 0.0085
Epoch 5/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0084 - val_
loss: 0.0084
Epoch 6/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0083 - val_
loss: 0.0082
Epoch 7/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0082 - val_
loss: 0.0081
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0081 - val_
loss: 0.0080
Epoch 9/10
1875/1875 [==============================] - 7s 3ms/step - loss: 0.0080 - val_
loss: 0.0081
Epoch 10/10
1875/1875 [==============================] - 7s 3ms/step - loss: 0.0080 - val_
loss: 0.0080
```

Out[105]:
```
<keras.callbacks.History at 0x7f9960accbd0>
```

Let's take a look at a summary of the encoder. Notice how the images are downsampled from 28x28 to 7x7.

In [106]:
```
autoencoder.encoder.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 14, 14, 16)        160

 conv2d_4 (Conv2D)           (None, 7, 7, 8)           1160

=================================================================
Total params: 1,320
Trainable params: 1,320
Non-trainable params: 0
_________________________________________________________________
```

The decoder upsamples the images back from 7x7 to 28x28.

In [107]:

```
autoencoder.decoder.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_transpose_2 (Conv2DT  (None, 14, 14, 8)        584       
 ranspose)                                                       
                                                                 
 conv2d_transpose_3 (Conv2DT  (None, 28, 28, 16)       1168      
 ranspose)                                                       
                                                                 
 conv2d_5 (Conv2D)           (None, 28, 28, 1)         145       
                                                                 
=================================================================
Total params: 1,897
Trainable params: 1,897
Non-trainable params: 0
_________________________________________________________________
```

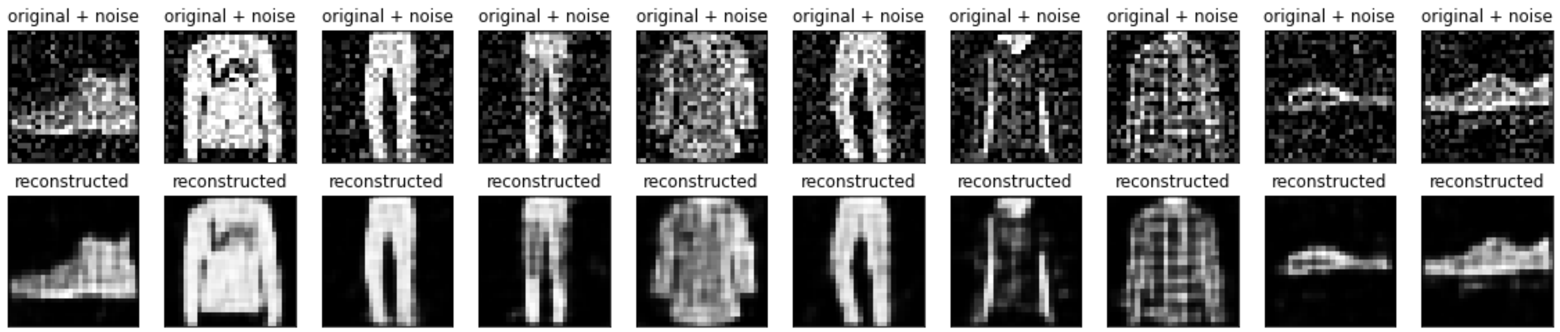
Plotting both the noisy images and the denoised images produced by the autoencoder.

In [108]:

```
encoded_imgs = autoencoder.encoder(x_test_noisy).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()
```

In [109]:

```
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):

    # display original + noise
    ax = plt.subplot(2, n, i + 1)
    plt.title("original + noise")
    plt.imshow(tf.squeeze(x_test_noisy[i]))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    bx = plt.subplot(2, n, i + n + 1)
    plt.title("reconstructed")
    plt.imshow(tf.squeeze(decoded_imgs[i]))
    plt.gray()
    bx.get_xaxis().set_visible(False)
    bx.get_yaxis().set_visible(False)
plt.show()
```



In [110]:
```
plt.savefig( os.path.join(tempdir, "autoencoder_denoise_example.png"))
```

```
<Figure size 432x288 with 0 Axes>
```

Third example: Anomaly detection

Overview

In this example, you will train an autoencoder to detect anomalies on the ECG5000 dataset (http://www.timeseriesclassification.com/description.php?Dataset=ECG5000). This dataset contains 5,000 Electrocardiograms (https://en.wikipedia.org/wiki/Electrocardiography), each with 140 data points. You will use a simplified version of the dataset, where each example has been labeled either `0` (corresponding to an abnormal rhythm), or `1` (corresponding to a normal rhythm). You are interested in identifying the abnormal rhythms.

Note: This is a labeled dataset, so you could phrase this as a supervised learning problem. The goal of this example is to illustrate anomaly detection concepts you can apply to larger datasets, where you do not have labels available (for example, if you had many thousands of normal rhythms, and only a small number of abnormal rhythms).

How will you detect anomalies using an autoencoder? Recall that an autoencoder is trained to minimize reconstruction error. You will train an autoencoder on the normal rhythms only, then use it to reconstruct all the data. Our hypothesis is that the abnormal rhythms will have higher reconstruction error. You will then classify a rhythm as an anomaly if the reconstruction error surpasses a fixed threshold.

Load ECG data

The dataset you will use is based on one from [timeseriesclassification.com (http://www.timeseriesclassification.com/description.php?Dataset=ECG5000)](http://www.timeseriesclassification.com/description.php?Dataset=ECG5000).

In [111]:

```
# Download the dataset
dataframe = pd.read_csv('http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv', header=None)
raw_data = dataframe.values
dataframe.head()
```

Out[111]:

	0	1	2	3	4	5	6	7	8	9	...	131	
0	-0.112522	-2.827204	-3.773897	-4.349751	-4.376041	-3.474986	-2.181408	-1.818286	-1.250522	-0.477492	...	0.792168	0.
1	-1.100878	-3.996840	-4.285843	-4.506579	-4.022377	-3.234368	-1.566126	-0.992258	-0.754680	0.042321	...	0.538356	0.
2	-0.567088	-2.593450	-3.874230	-4.584095	-4.187449	-3.151462	-1.742940	-1.490659	-1.183580	-0.394229	...	0.886073	0.
3	0.490473	-1.914407	-3.616364	-4.318823	-4.268016	-3.881110	-2.993280	-1.671131	-1.333884	-0.965629	...	0.350816	0.
4	0.800232	-0.874252	-2.384761	-3.973292	-4.338224	-3.802422	-2.534510	-1.783423	-1.594450	-0.753199	...	1.148884	0.

5 rows × 141 columns

In [112]:
```
# The last element contains the labels
labels = raw_data[:, -1]

# The other data points are the electrocadriogram data
data = raw_data[:, 0:-1]

train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size=0.2, random_state=21
)
```

Normalize the data to `[0,1]`.

In [113]:
```
min_val = tf.reduce_min(train_data)
max_val = tf.reduce_max(train_data)

train_data = (train_data - min_val) / (max_val - min_val)
test_data = (test_data - min_val) / (max_val - min_val)

train_data = tf.cast(train_data, tf.float32)
test_data = tf.cast(test_data, tf.float32)
```

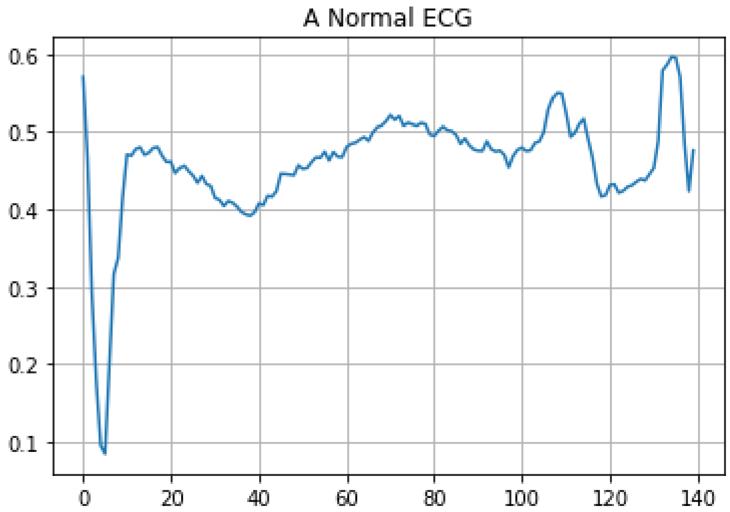
You will train the autoencoder using only the normal rhythms, which are labeled in this dataset as `1`. Separate the normal rhythms from the abnormal rhythms.

In [114]:
```
train_labels = train_labels.astype(bool)
test_labels = test_labels.astype(bool)

normal_train_data = train_data[train_labels]
normal_test_data = test_data[test_labels]

anomalous_train_data = train_data[~train_labels]
anomalous_test_data = test_data[~test_labels]
```

Plot a normal ECG.

In [115]:

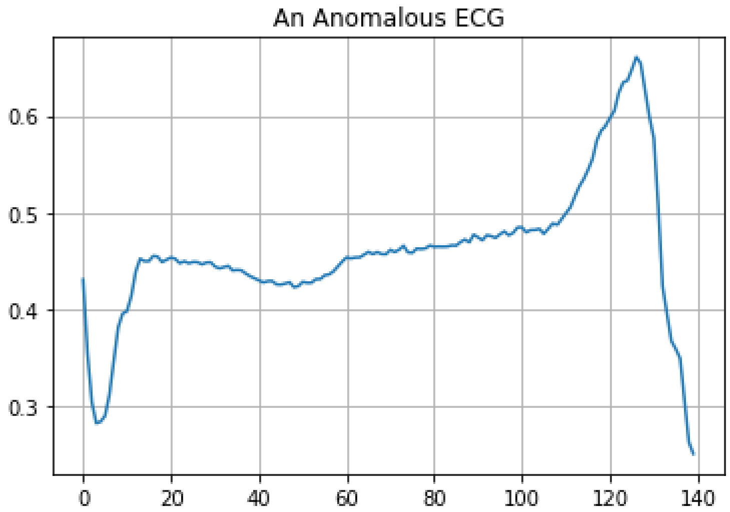
```
plt.grid()
plt.plot(np.arange(140), normal_train_data[0])
plt.title("A Normal ECG")

plt.savefig( os.path.join(tempdir, "autoencoder_anomaly_normal.png"))
plt.show()
```



Plot an anomalous ECG.

In [116]:

```
plt.grid()
plt.plot(np.arange(140), anomalous_train_data[0])
plt.title("An Anomalous ECG")

plt.savefig( os.path.join(tempdir, "autoencoder_anomaly_anomalous.png"))
plt.show()
```



Build the model

In [117]:

```
class AnomalyDetector(Model):
  def __init__(self):
    super(AnomalyDetector, self).__init__()
    self.encoder = tf.keras.Sequential([
      layers.Dense(32, activation="relu"),
      layers.Dense(16, activation="relu"),
      layers.Dense(8, activation="relu")])
```