In [61]:
```python
@interact(image_idx=(0, 1000))
def show_num_components_reconst(image_idx):
    fig, ax = plt.subplots(figsize=(20., 20.))
    actual = X[image_idx]
    # concatenate the actual and reconstructed images as large image before plotting it
    x = np.concatenate([actual[np.newaxis, :], reconstructions[:, image_idx]])
    ax.imshow(np.hstack(x.reshape(-1, 28, 28)[np.arange(10)]),
              cmap='gray');
    ax.axvline(28, color='orange', linewidth=2)
```

```
interactive(children=(IntSlider(value=500, description='image_idx', max=1000), Output()), _dom_classes=('widge…
```

We can also browse through the reconstructions for other digits. Once again, `interact` becomes handy for visualing the reconstruction.

In [62]:
```python
@interact(i=(0, 10))
def show_pca_digits(i=1):
    """Show the i th digit and its reconstruction"""
    plt.figure(figsize=(4,4))
    actual_sample = X[i].reshape(28,28)
    reconst_sample = (reconst[i, :]).reshape(28, 28)
    plt.imshow(np.hstack([actual_sample, reconst_sample]), cmap='gray')
    plt.grid(False)
    plt.show()
```

```
interactive(children=(IntSlider(value=1, description='i', max=10), Output()), _dom_classes=('widget-interact',…
```

PCA for high-dimensional datasets

Sometimes, the dimensionality of our dataset may be larger than the number of samples we have. Then it might be inefficient to perform PCA with your implementation above. Instead, as mentioned in the lectures, you can implement PCA in a more efficient manner, which we call "PCA for high dimensional data" (PCA_high_dim).

Below are the steps for performing PCA for high dimensional dataset

1. Normalize the dataset matrix X to obtain $\overline{X}$ that has zero mean.
2. Compute the matrix $\overline{X}\overline{X}^T$ (a N by N matrix with $N << D$)
3. Compute eigenvalues λs and eigenvectors V for $\overline{X}\overline{X}^T$ with shape (N, N). Compare this with computing the eigenspectrum of $\overline{X}^T\overline{X}$ which has shape (D, D), when $N << D$, computation of the eigenspectrum of $\overline{X}\overline{X}^T$ will be computationally less expensive.
4. Compute the eigenvectors for the original covariance matrix as $\overline{X}^T V$. Choose the eigenvectors associated with the `n` largest eigenvalues to be the basis of the principal subspace U.
   A. Notice that $\overline{X}^T V$ would give a matrix of shape (D, N) but the eigenvectors beyond the Dth column will have eigenvalues of 0, so it is safe to drop any columns beyond the D'th dimension.
   B. Also note that the columns of U will not be unit-length if we pre-multiply V with $\overline{X}^T$, so we will have to normalize the columns of U so that they have unit-length to be consistent with the `PCA` implementation above.
5. Compute the orthogonal projection of the data onto the subspace spanned by columns of U.

Functions you wrote for earlier assignments will be useful.

In [75]:
```python
# GRADED FUNCTION: DO NOT EDIT THIS LINE
def PCA_high_dim(X, num_components):
    """Compute PCA for small sample size but high-dimensional features.
    Args:
        X: ndarray of size (N, D), where D is the dimension of the sample,

           and N is the number of samples
        num_components: the number of principal components to use.
    Returns:
        X_reconstruct: (N, D) ndarray. the reconstruction
        of X from the first `num_components` pricipal components.
    """
    # YOUR CODE HERE
    # Uncomment and modify the code below
    N, D = X.shape
#     # Normalize the dataset
    X_normalized, mean = normalize(X)
#     # Find the covariance matrix
    M = np.dot(X_normalized, X_normalized.T) / N

#     # Next find eigenvalues and corresponding eigenvectors for S
#     # Make sure that you only take the first D eigenvalues/vectors
#     # You can also take a look at the eigenvalues beyond column (D-1) and they should be
#     # zero (or a very small number due to finite floating point precision)
    eig_vals, eig_vecs = eig(M)

#     # Compute the eigenvalues and eigenvectors for the original system
#     # eig_vecs = None

#     # Normalize the eigenvectors to have unit-length
#     # Take the top `num_components` of the eigenvalues / eigenvectors
#     # as the principal values and principal components
    principal_values = eig_vals[:num_components]
    principal_components = eig_vecs[:, :num_components]

#     # Due to precision errors, the eigenvectors might come out to be complex, so only take their real parts
    principal_components = np.real(principal_components)

#     # reconstruct the images from the lower dimensional representation
#     # Remember to add back the sample mean
    reconst = (projection_matrix(principal_components)@ X_normalized) + mean
    return reconst, mean, principal_values, principal_components
```

In [76]:
```
# YOUR CODE HERE
```

In [77]:
```
# Some hidden tests below
### ...
```

Given the same dataset, `PCA_high_dim` and `PCA` should give the same output. Assuming we have implemented `PCA`, correctly, we can then use `PCA` to test the correctness of `PCA_high_dim`. Given the same dataset, `PCA` and `PCA_high_dim` should give identical results.

We can use this **invariant** to test our implementation of PCA_high_dim, assuming that we have correctly implemented `PCA`.

In [78]:
```
random = np.random.RandomState(0)
# Generate some random data
X = random.randn(5, 4)
pca_rec, pca_mean, pca_pvs, pca_pcs = PCA(X, 2)
pca_hd_rec, pca_hd_mean, pca_hd_pvs, pca_hd_pcs = PCA_high_dim(X, 2)
# Check that the results returned by PCA and PCA_high_dim are identical
np.testing.assert_allclose(pca_rec, pca_hd_rec)
np.testing.assert_allclose(pca_mean, pca_hd_mean)
np.testing.assert_allclose(pca_pvs, pca_pvs)
np.testing.assert_allclose(pca_pcs, pca_pcs)
```

Congratulations! You have now learned how PCA works!