WORLDQUANT UNIVERSITY

Compiled Content Module 6

MScFE 650

Machine Learning in Finance

Revised: 09/07/2019

Table of Contents

Module 6: Deep Learning Sequential Models

In Module 6, we build on what was covered in the previous module, beginning with the algorithm used to optimize the parameters of a neural network – namely, backpropagation. Next, we turn to convolutional and recurrent neural networks, after which you are given an opportunity to hone your skills with coding examples. The module then concludes with a brief look at dropout.

Unit 1: Backpropagation

Any gradient-based optimization method needs the gradient. Moreover, if the gradient is available, then gradient-based methods tend to be fast. The fact that it is indeed possible to calculate gradients efficiently probably goes a long way towards explaining the current popularity of gradient-based methods for training neural networks. This algorithm, known as **backpropagation**, is the topic of this notebook.

There is nothing particularly deep about backpropagation. In essence, it is just an efficient way of calculating the chain rule (calculus). The basic ideas are easily explained by means of a simple example.

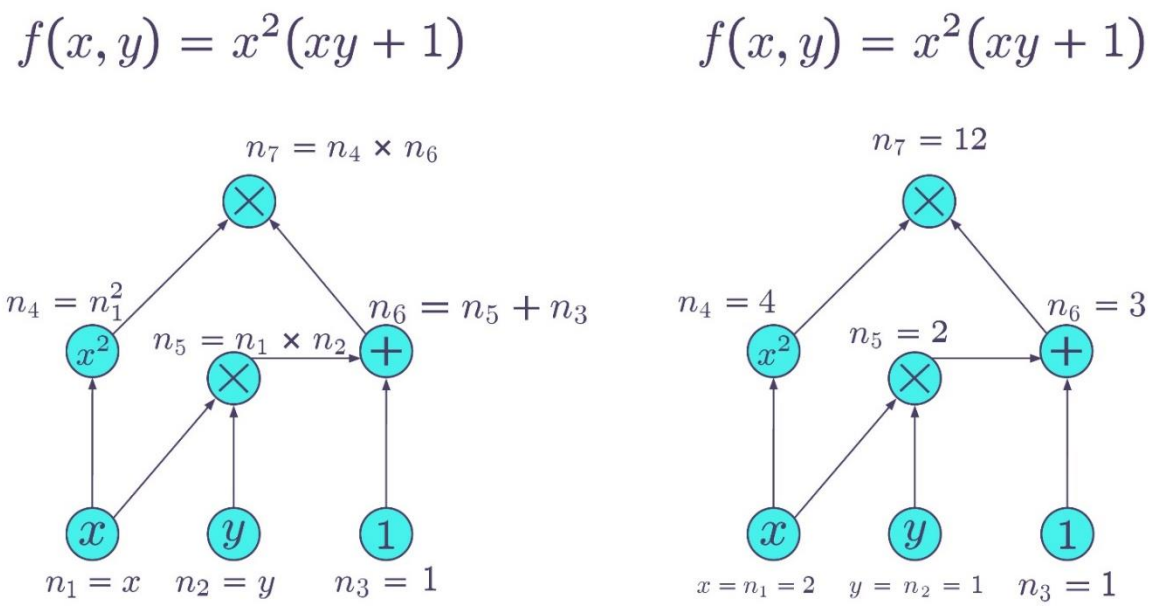


Figure 3: The forward pass

On the left side of Figure 3, the computational graph of the function $f(x,y) = x^2(xy+1)$ is shown. Note that we also label the computational nodes from n_1 to n_7, where n_7 is simply the value of f. The graph shows the flow from the input nodes $n_1 = x, n_2 = y$, and $n_3 = 1$, up to the output node n_7, which is just f, as mentioned above. You can think of this as a symbolic graph illustrating the connections between the different nodes. On the right side, we see how actual input data, $x = 2, y = 1,$ and $n_3 = 1,$ flows through the graph. The values of all the nodes are saved for future use, more specifically, during the backward pass.

The goal is to calculate the gradient,

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}.$$

For this, we clearly need to calculate the two partial derivatives. If we do it ourselves, we'll simply apply the chain rule to obtain the answers. Backpropagation does this in a systematic manner.

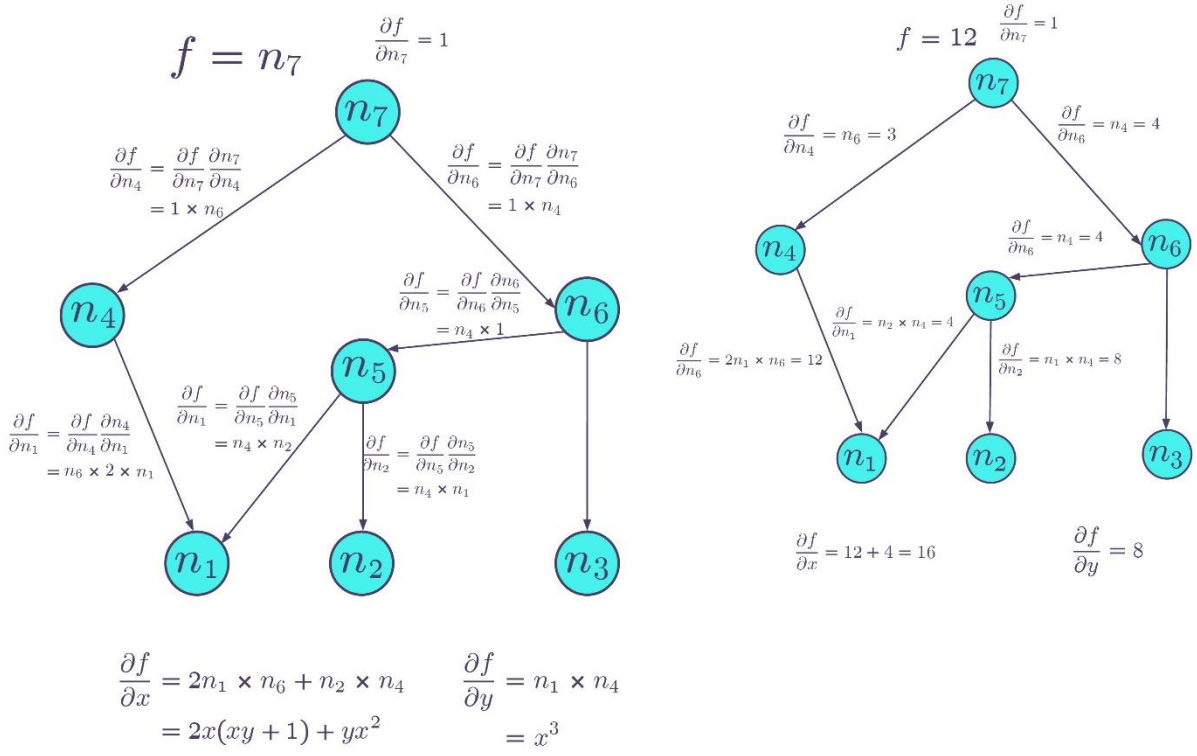


Figure 4: The backward pass

Since the partial derivatives of f, with respect to x and y, are required, we proceed from the top. Since $f = n_7$, it follows that $\frac{\partial f}{\partial n_7} = 1$. From n_7, both branches should be followed, as indicated in the figure on the left. The first thing to note is that all the nodal values that we are required to calculate are already available from the forward pass. It is therefore straightforward to calculate all the partial derivatives with respect to all the nodes, systematically, starting at the top output node. Working our way "backwards" through the figure, we eventually arrive at x via two different paths. The partial derivative with respect to x is simply the sum of the partial derivatives along the two paths. Since y is reached from only one path, the partial derivative, as calculated along that path, is the required partial derivative.

Fully-connected neural network

In order to illustrate how this can be made into a systematic algorithm, we'll now consider a fully-connected neural network with activation functions $\sigma(\cdot)$.

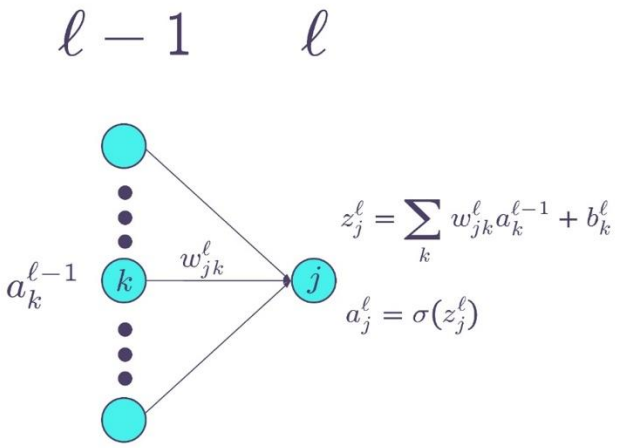


Figure 5: Connection between two layers

Keep in mind that σ is a general activation function, not necessarily the sigmoid function, and that it can even be a different function for each layer. All that is important is that we'll be able to calculate its derivative $\sigma'(\cdot)$. It may be worth pointing out that σ is applied pointwise on vectors.

If

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}, \text{ then } \sigma(\mathbf{x}) \text{ is a vector with components } \sigma(\mathbf{x}) = \begin{bmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_d) \end{bmatrix}.$$

Figure 5 shows how information from layer $\ell - 1$ is passed to layer ℓ. The output of the top layer, given an input vector $\mathbf{x}$, is a vector $\mathbf{a}^L(\mathbf{x})$. The mean cost function is then given by the average over all input vectors,

$$C(W) = \frac{1}{N}\sum_x C_x(W),$$

where $C_x(W)$ is the cost function for a single input vector, and W indicates all the network parameters. Thus, $C_x(W)$ is a function of the model output $\mathbf{a}^L(\mathbf{x})$ and the targets t, not indicated in the notation.

Notes

- The expression of the average cost function is, of course, an assumption that it can be written as the mean over individual input vectors.
- Examples of often-used cost functions of this form include mean-squared error and cross-entropy.

We need to calculate the gradient of $C(W)$ with respect to all the parameters. Note that this is just the sum of the gradients for the individual input vectors $C_{\mathrm{x}}(W)$. More specifically, we need to calculate the gradients with respect to all the offsets, $\frac{\partial C}{\partial b_k^\ell}$ and all the weights $\frac{\partial C}{\partial w_{jk}^\ell}$ where we have now dropped the dependencies on **x** and W in the notation. Also note that w_{jk}^ℓ is the weight connecting the k^{th} neuron in layer $\ell-1$ to the j^{th} neuron in the ℓ^{th} layer, as shown in the figure. Referring to the figure above, we can therefore write

$$\begin{aligned}\frac{\partial C}{\partial w_{jk}^\ell} &= \frac{\partial C}{\partial z_j^\ell}\frac{\partial z_j^\ell}{\partial w_{jk}^\ell} \\ &= \delta_j^\ell a_k^{\ell-1},\end{aligned}$$

where we somewhat ambiguously refer to $\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell}$ as the "error". The partial derivatives with respect to the offsets are given by,

$$\begin{aligned}\frac{\partial C}{\partial b_j^\ell} &= \frac{\partial C}{\partial z_j^\ell}\frac{\partial z_j^\ell}{\partial b_j^\ell} \\ &= \delta_j^\ell.\end{aligned}$$

Note: The values of the activations a_k^ℓ are calculated during the forward sweep and stored in memory. Assuming, of course, that enough memory is available.

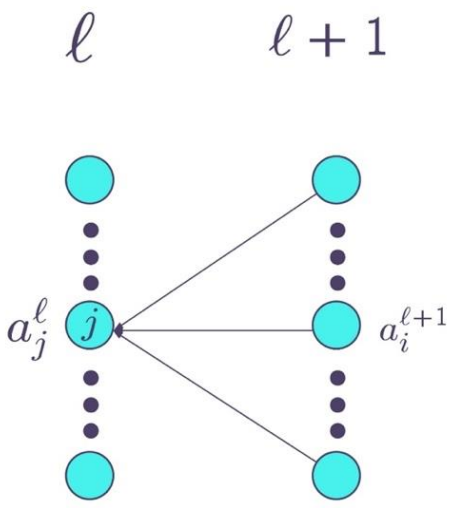


Figure 6: Connection between layer $\ell + 1$ and layer ℓ

Let's see how the errors propagate through the network, starting from the output layer,

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

$$= \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$

$$= \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L).$$

If δ_j^ℓ can be calculated from the $\delta_i^{\ell+1}$ of the upper layer, we have a recursion, and an algorithm for calculating the gradients.

We start with

$$\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell}$$

$$= \sum_i \frac{\partial C}{\partial z_i^{\ell+1}}\frac{\partial z_i^{\ell+1}}{\partial z_j^\ell}$$

$$= \sum_i \delta_i^{\ell+1}\frac{\partial z_i^{\ell+1}}{\partial z_j^\ell}.$$

Since

$$z_i^{\ell+1} = \sum_k w_{ik}^{\ell+1} a_k^{\ell} + b_i^{\ell+1}$$

$$= \sum_k w_{ik}^{\ell+1} \sigma(z_k^{\ell}) + b_i^{\ell+1},$$

it follows that

$$\frac{\partial z_i^{\ell+1}}{\partial z_j^{\ell}} = w_{ij}^{\ell+1} \sigma'(z_j^{\ell}),$$

and

$$\delta_j^{\ell} = \sum_i \delta_i^{\ell+1} w_{ij}^{\ell+1} \sigma'(z_j^{\ell}).$$

This allows a recursion. Starting from the top, we calculate the δ's through the lower layers, all the way down to the bottom layer. Since we have all the necessary values in memory from the forward sweep, this is fast. Moreover, from the δ's, we calculate all the necessary partial derivatives.

The figure below gives a schematic overview of how the different layers fit together.

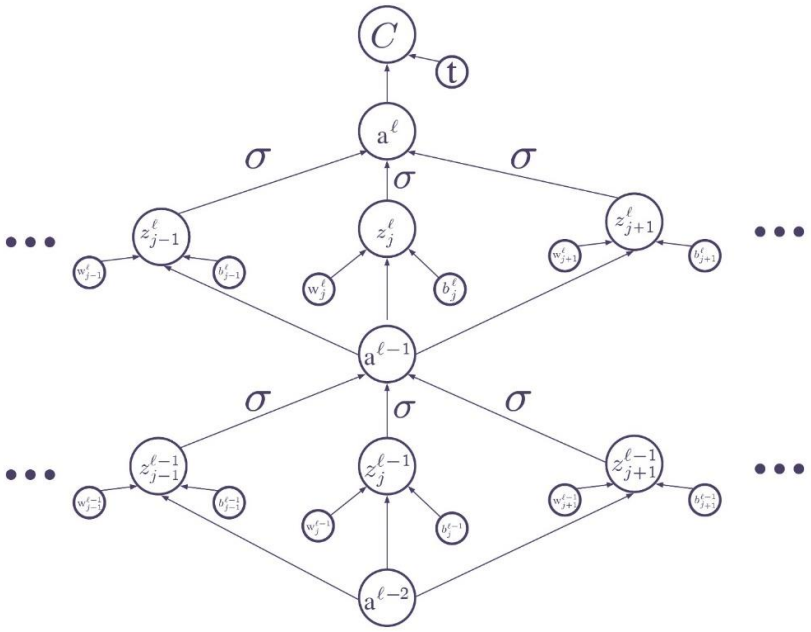


Figure 7: How the layers fit together

We are now in a position to indicate why it is hard to train deep networks. First note that the update in a parameter is proportional to its component in the gradient. To be more specific, the update in w_{jk}^{ℓ} is proportional to $\frac{\partial C}{\partial w_{jk}^{\ell}}$ given by,

$$\frac{\partial C}{\partial w_{jk}^{\ell}} = \delta_j^{\ell} a_k^{\ell-1}.$$

If this quantity is small, the update is small, and it may take a long time to train the parameter. Thus, if either $a_k^{\ell-1}$ or δ_j^{ℓ} is small, training will slow down. The latter depends on the derivative of the activation function, $\sigma'(z_j^{\ell})$. The sigmoid activation function quickly saturates as the logits move away from 0, slowing down the training. Moreover, the values in the different layers are essentially multiplied, meaning that the gradient can become very small just before the input layer. But this is exactly where the information is fed into the network.

On the other hand, if the derivative of the activation function is larger than 1, then this value is multiplied as backpropagation passes through the different layers. This can lead to exploding gradients.

It is clearly important to avoid both vanishing and exploding gradients through a judicious choice of the activation function, among other considerations.

Unit 2: Convolutional Neural Networks

One of the problems with using deep neural networks is the large number of parameters that need to be trained. It requires a significant amount of training data, and then there is always the problem of overfitting. This is especially the case when we use images as input. Even a low-resolution image – say, 28 by 28 – will already have 784 dimensions. Another problem is that, whenever the resolution of the image changes, the input dimension changes, and the neural network is no longer applicable. Convolutional neural networks (CNN's) come to the rescue.

Biological origin

CNN's have an interesting biological origin, as established by the Nobel Prize-winning research of Hubert and Wiesel. Quoting from the aforementioned webpage: "The classic experiments by Hubel and Wiesel are fundamental to our understanding of how neurons along the visual pathway extract increasingly complex information from the pattern of light cast on the retina to construct an image" (Fehlhaber, 2014). This is the process that we try to replicate with CNN's. A local filter is created that responds in the same way to similar stimuli in different parts of the image. By stacking multiple filters and neural layers, the CNN is capable of extracting increasingly complex information from the image.

Also note the reduction in the number of trainable coefficients. If we have a 3 by 3 filter, we have only 9 unknown coefficients that we need to estimate during training. If we reduce the image, say a 28 by 28 image in the usual way, to a 784-dimensional vector, 784 unknown coefficients need to be trained. Of course, the savings is not quite as large since we'll use several small filters, but in general can be substantial.

Convolution

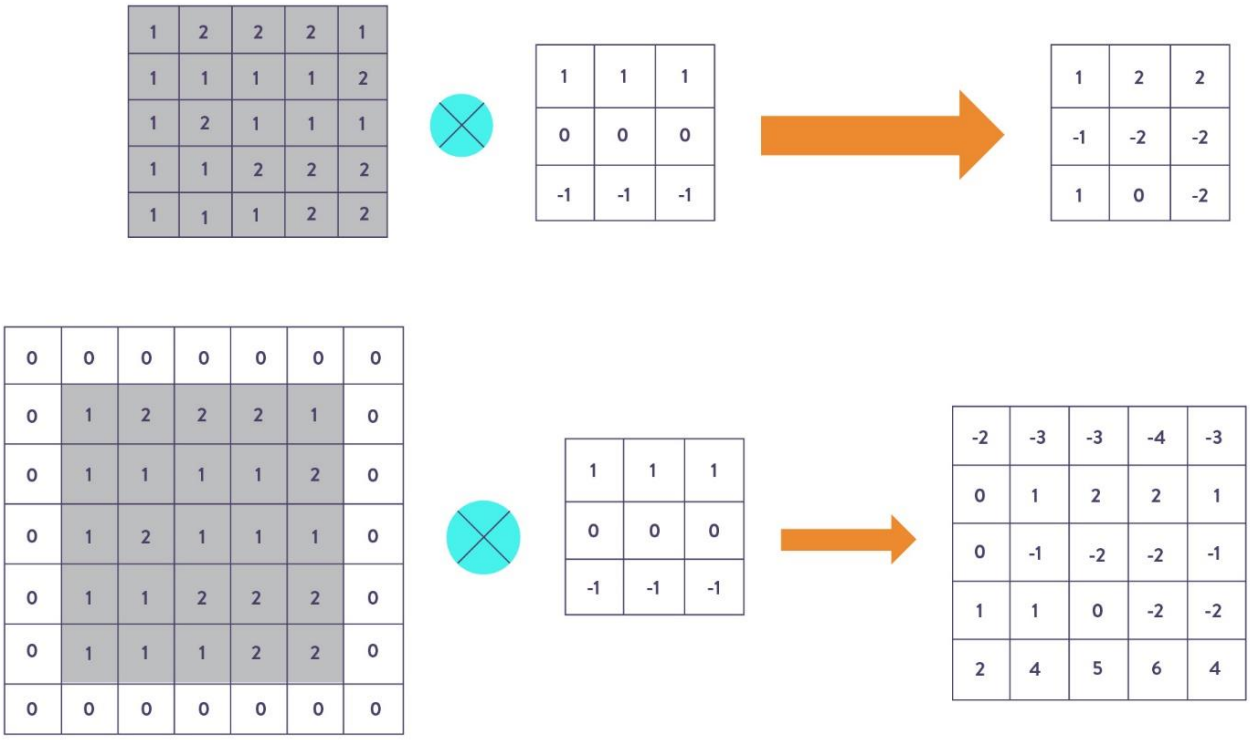


Figure 10: Convolution

The above figure illustrates the idea of convolution as used in images. The attentive reader will note that it differs slightly from the usual definition. On the top left a 5-by-5 grayscale image is shown. This is convolved with a 3-by-3 filter, shown to its right. The way it works is that the filter is placed on top of the image to cover a 3-by-3 patch of the image. The filter coefficients are then multiplied with the corresponding pixel values underneath the filter. These values are summed to produce a single output value – a pixel value of the new image. The filter is then moved one position over (in this example), and the process is repeated.

It is immediately apparent that we have a problem at the edges. If we fit the filter inside the original image, the resulting image is smaller. In this case it is reduced from a 5-by-5 image to a 3-by-3 image. This may not matter much but, if the same-sized output image as the original image is required, it is necessary to do something at the boundary. One frequently used technique is to pad the original images with zeros, as shown in the bottom-left image. The filter is then applied as before, but, in this case, the output image dimension is the same as before.

There are several issues the reader should note:

- We showed a 3-by-3 filter but obviously this is another hyperparameter that the user has to control. Larger filter sizes cover more of the image, and therefore take more context into account. Nowadays, the trend is to use smaller filter sizes.
- The amount of padding clearly depends on the filter size. Larger filter sizes require more padding.
- We moved the filter one step at a time. This is known as the **stride** and is also a hyperparameter. For larger filters, a larger stride can be considered.
- Instead of outputting a single image, this is not what is done in practice, since capturing all the relevant information does not provide sufficient "richness" to the system. Instead, a whole stack of filters is used – perhaps as deep as 32 different filters. Each filter is applied to the image as before, and each filter outputs an image as before. Again, this is a hyperparameter that the user should apply.
- In this example, we started with a single image; but if we were given a color image, we would have thought of that as three different images, each corresponding to one of the three primary colors in images: red, green, and blue. The filter is applied to the whole stack of images; all the values are multiplied and added. This is, of course, particularly pertinent if the output of a convolutional layer consists of a stack of multiple images.
- During training, the filter coefficients need to be calculated. If we specify 3-by-3 filters and a stack of 32 – a total of 3-by-3-by-32 coefficients need to be calculated.

In summary, the user needs to supply the following information (Note that different implementations in TensorFlow need different inputs – e.g., `tf.nn.conv2d` and `tf.layers.conv2d`):

- The number of filters,
- Kernel size (size of the filter),
- Stride. (Strides $= [x, y]$),
- The size of the output image – e.g., whether it needs to be the same as the original, in which case padding is added. Padding can therefore be VALID or SAME.

Pooling layer

Convolutional layers are often followed by a so-called pooling layer. The purpose is to reduce the image size, and also provide for a limited amount of translation invariance.

As with CNN's, a filter of a specific size is selected. This is then placed on top of the original image or images. However, unlike CNN's, all the pixel values underneath the pooling filter are now replaced by a single value. This is again something that needs to be specified and two popular choices are:

- Replacement by the maximum, in which case it is unsurprisingly known as **max pooling**.
- Replacement by the average, known as **average pooling**.
- You can think of pooling as a filter of size = [batch_size, height, width, channels]. Since this is not fully supported, use size = [1, height, width, 1].
- You also have to set strides = [1, x, y, 1].

We now illustrate how this is used in practice:

In [1]:
```
%matplotlib inline
import tensorflow as tf
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.cm as cm
```

Illustration of convolutional filters

In this first illustration, we simply investigate the output of a convolution filter as applied to a sample image.

We first load an image from the scikit-learn dataset:

In [2]:
```
from sklearn.datasets import load_sample_image

china = load_sample_image("china.jpg")
image = china[150:220, 130:250]
height, width, channels = image.shape
image_grayscale = image.mean(axis=2).astype(np.float32)
images = image_grayscale.reshape(1, height, width, 1)
```

```
C:\Users\User\Anaconda2\envs\wqu_ml_fin\lib\site-packages\sklearn\datasets\base.py:762: DeprecationWarning: `imread` is deprec
ated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  images = [imread(filename) for filename in filenames]
C:\Users\User\Anaconda2\envs\wqu_ml_fin\lib\site-packages\sklearn\datasets\base.py:762: DeprecationWarning: `imread` is deprec
ated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  images = [imread(filename) for filename in filenames]
```

You can ignore any depreciation warnings for now. Python packages are constantly updated so it's hard to write material where the functions you use will always be the functions that exist. To combat this, we ask students to install specific versions of a package.

In [18]:
```
# The original color image
plt.imshow(image)
plt.title('Color Image')
plt.show()
```

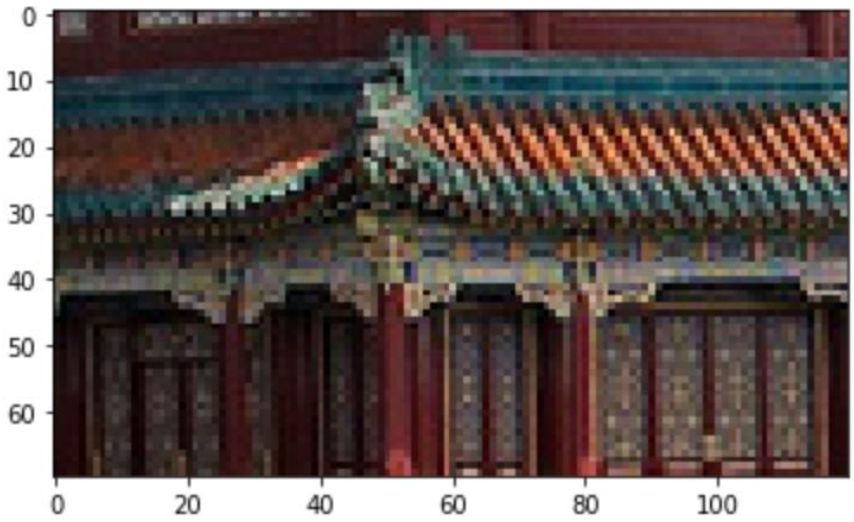


Figure 10: Color image

In [19]:
```
# See what the image looks like in grayscale
plt.imshow(images[0, :, :, 0], cmap="gray")
plt.title('Grayscale')
plt.show()
```

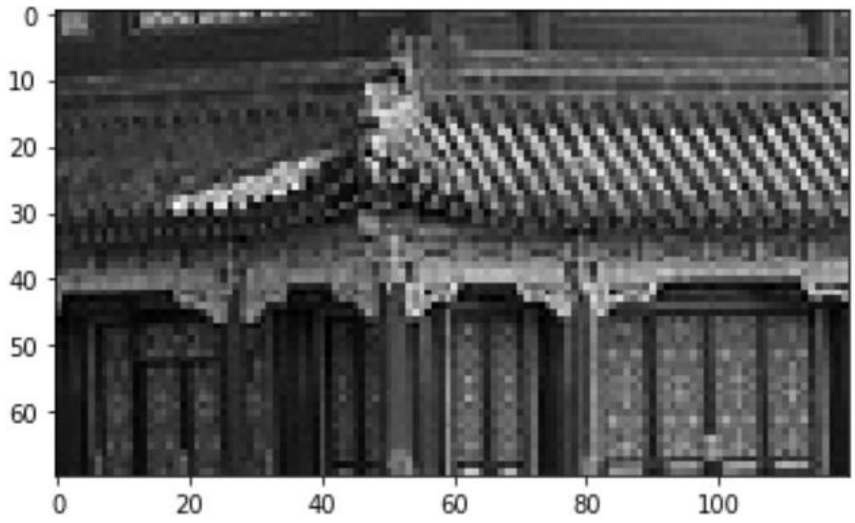


Figure 11: Grayscale image

Filters

Next, we construct two filters that are sensitive to horizontal and vertical features in the image.

In [20]:
```
# Create 2 filters
# The shape of the  filter is [height, width, channels, filters]

fmap = np.zeros(shape=(7, 7, 1, 2), dtype=np.float32)
fmap[:, 3, 0, 0] = 1 # first filter
fmap[3, :, 0, 1] = 1 # second filter

plt.imshow(fmap[:, :, 0, 0], cmap="gray", interpolation="nearest")
plt.title('Vertical Filter')
plt.show()

plt.imshow(fmap[:, :, 0, 1], cmap="gray", interpolation="nearest")
plt.title('Horizontal Filter')
plt.show()
```

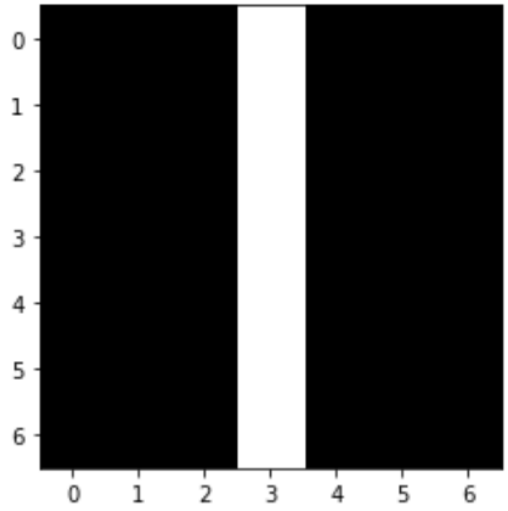


Figure 12: Vertical filter

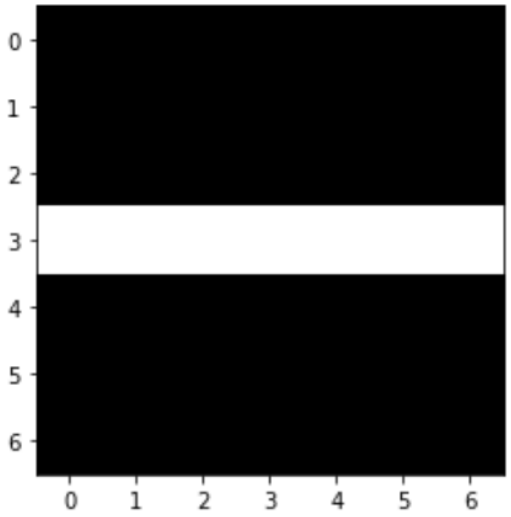


Figure 13: Horizontal filter

Above, we can see that we have two 7-by-7 filters. One is a vertical filter, and the other a horizontal filter.

Now, we apply the two filters to the grayscale image and then look at the resulting two new images – the first, which is passed through a horizontal filter, and the second, a vertical filter. Note how we make use of "SAME" padding to retain the image dimensions.

In [21]:
```
tf.reset_default_graph()

X = tf.placeholder(tf.float32, shape=(None, height, width, 1))
feature_maps = tf.constant(fmap)
convolution = tf.nn.conv2d(X, feature_maps, strides=[1, 1, 1, 1], padding="SAME")

with tf.Session() as sess:
    output = convolution.eval(feed_dict={X: images})
```

In [22]:
```
plt.imshow(output[0, :, :, 1], cmap="gray")
plt.title('Response to Horizontal Features')
plt.show()
```

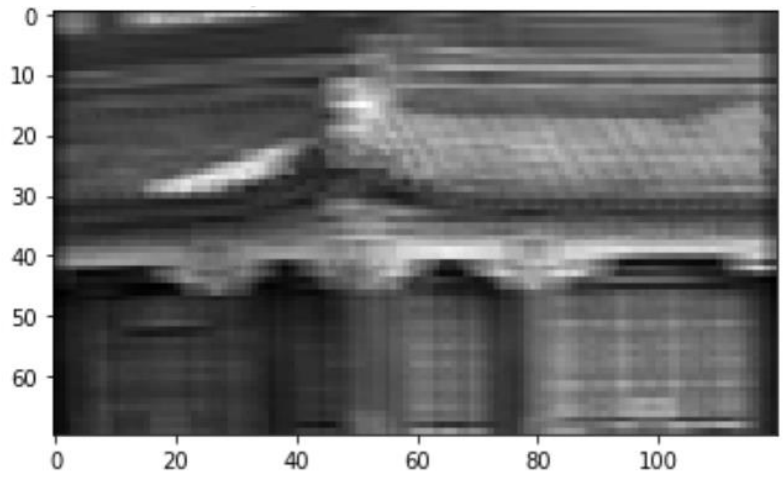


Figure 14: Response to horizontal features

In [23]:
```
plt.imshow(output[0, :, :, 0], cmap="gray")
plt.title('Response to Vertical Features')
plt.show()
```

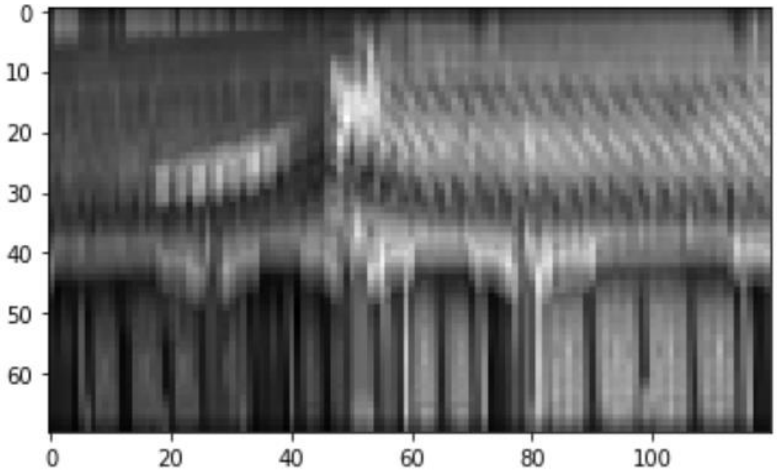


Figure 15: Response to vertical features

Now you can see the effect that the different filters have – for example, the vertical filter has left us only with an image composed of vertical lines. Next, let's move on to an example using the familiar MNIST data.

MNIST classification using convolutional neural networks

The construction uses `tf.layers.conv2`. A few points to note:

1. The default initialization is rather efficient, but the details are not documented. It is possible to choose your own initialization.
2. The bias is added by default. The documentation is misleading.
3. The trainable variables are not explicitly named in `tf.layers`. In order to fetch their values, one has to extract them from the default graph. (Note the example provided.)
4. Dropout is only used during training, and not during testing. The easiest way to do this is to define `keep_prob` as a placeholder. During testing, `keep_prob` is set to 1.

Import the data

The MNIST data is available in TensorFlow and provides a useful helper-function for feeding it in mini-batches.

In [24]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

print()
print(type(mnist.train.images))
print(mnist.train.images.shape)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz

<class 'numpy.ndarray'>
(55000, 784)
```

Construction phase

As usual, we start with the construction phase. The MNIST images are 28 by 28 and are grayscale, so they only have 1 channel.

We will have 32 different filters, each of size 3 by 3, with a stride of 1. We also add “SAME” padding to ensure that the resulting image is of the same size. Next, a max-pooling layer is added and then a standard fully connected layer (feed forward neural network) with 64 nodes that feed into a SoftMax with 10 outputs. Dropout is also added in the fully connected layer.

In [25]:
```
tf.reset_default_graph()

# The MNIST images
height = 28
width = 28
channels = 1
n_inputs = height*width

# First convolutional layer
conv1_fmaps = 32
conv1_ksize = 3
conv1_stride = 1
conv1_pad = "SAME"

# Max Pooling layer
pool2_fmaps = conv1_fmaps

#Fully connected, and output layer
n_fc1 = 64
n_outputs = 10


with tf.name_scope("Inputs"):
    X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
    X_reshaped = tf.reshape(X, shape=[-1, width, height, channels])
    y = tf.placeholder(tf.float32, shape=(None, n_outputs), name="y")
    keep_prob = tf.placeholder(tf.float32, name="keep_proba")

with tf.name_scope("Conv1"):
    conv1 = tf.layers.conv2d(X_reshaped, filters=conv1_fmaps, kernel_size=conv1_ksize,
                         strides=conv1_stride, padding=conv1_pad,
                         activation=tf.nn.relu, name="conv1")

with tf.name_scope("pool2"):
    pool2 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="VALID")
    pool2_flat = tf.reshape(pool2, shape=[-1, pool2_fmaps * 14 * 14])

with tf.name_scope("fc1"):
    fc1 = tf.layers.dense(pool2_flat, n_fc1, activation=tf.nn.relu, name="fc1")

with tf.name_scope("dropout"):
    fc11 = tf.nn.dropout(fc1, keep_prob=keep_prob )

with tf.name_scope("output"):
    logits = tf.layers.dense(fc1, n_outputs, name="output")
    Y_prob = tf.nn.softmax(logits, name="Y_prob")

with tf.name_scope("train"):
    xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y, name="xentropy")
    loss = tf.reduce_mean(xentropy, name='loss')
    optimizer = tf.train.AdamOptimizer()
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.equal(tf.argmax(logits, axis=1), tf.argmax(y, axis=1))
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

with tf.name_scope("init_and_save"):
    init_op = tf.global_variables_initializer()
    saver = tf.train.Saver()
```

Execution phase

Start training

Now we run the execution phase to train the model on the data. Please note that this can take a while (about 10 minutes), so please be patient.

The model is trained for 10 epochs, with a batch size of 50. Additionally, the dropout rate is set to 50% (a standard value).

In [26]:
```
n_epochs = 10
batch_size = 50

with tf.Session() as sess:
    sess.run(init_op)
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch, keep_prob: 0.5})
        acc_train = accuracy.eval(feed_dict={X: mnist.train.images, y: mnist.train.labels, keep_prob: 1.0})
        acc_test  = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels, keep_prob: 1.0})
        print(epoch, "Train accuracy:", acc_train, ",  Test accuracy:", acc_test)

    var = [v.name for v in tf.trainable_variables()]
    print(var)

    # Fetch the variables from the graph
    graph = tf.get_default_graph()
    filters = graph.get_tensor_by_name("conv1/kernel:0").eval()
    bias    = graph.get_tensor_by_name("conv1/bias:0").eval()

    # Save the graph and the current values of the variables.
    save_path = saver.save(sess,"./mnist_cnn/mnist.ckpt")
```

```
0 Train accuracy: 0.9811636 ,  Test accuracy: 0.9779
1 Train accuracy: 0.9874 ,  Test accuracy: 0.9831
2 Train accuracy: 0.99 ,  Test accuracy: 0.9831
3 Train accuracy: 0.9922909 ,  Test accuracy: 0.9836
4 Train accuracy: 0.9964727 ,  Test accuracy: 0.987
5 Train accuracy: 0.9964727 ,  Test accuracy: 0.9841
6 Train accuracy: 0.9971091 ,  Test accuracy: 0.9845
7 Train accuracy: 0.9960727 ,  Test accuracy: 0.9841
8 Train accuracy: 0.9976364 ,  Test accuracy: 0.9854
9 Train accuracy: 0.9962364 ,  Test accuracy: 0.9838
['conv1/kernel:0', 'conv1/bias:0', 'fc1/kernel:0', 'fc1/bias:0', 'output/kernel:0', 'output/bias:0']
```

The model has been trained and you should have a test accuracy above 98%. If we want to have a look at the variables from the graph, we can do so below.

In [27]:
```
print('filters:', filters.shape, 'bias:', bias.shape)
print('bias:',bias)
```

```
filters: (3, 3, 1, 32) bias: (32,)
bias: [-0.21391982 -0.00094368 -0.22411765 -0.01227435 -0.08120982 -0.2032379
 -0.18339497 -0.01951492 -0.14985445 -0.20981443 -0.20136009 -0.14826804
 -0.06141082 -0.16475852 -0.05373911 -0.09444895 -0.1002607  -0.13797547
 -0.12806681 -0.02573098 -0.20530017 -0.00573664 -0.27695793 -0.22163717
 -0.01238952 -0.14137559 -0.07772201 -0.05206067 -0.06555817 -0.17443494
 -0.04171394 -0.00393423]
```

Unit 3: Recurrent Neural Networks

It is time to break out of the confinement of the feedforward neural networks studied up until now. We now move on to recurrent neural networks (RNN's). Please note that, for this, ChrisOlah's blog, Understanding LSTMs is compulsory reading.

The problem with feedforward networks is that they cannot naturally handle dynamic systems, such as time series. They require a fixed-length input vector, and after a fixed number of steps, produce a fixed-size output. In contrast, RNN's allow us to operate on sequences of vectors, and it is not hard to find important applications for this idea.

In finance, we observe the markets in all their volatility and we want to predict the future, taking history into account. In natural language processing, one may want to translate a sentence in one language into its equivalent in another language. In this case, the input is a sequence of words in one language and the output is another sequence of words, the translated text.

There is something miraculous about RNN's. In Karpathy (2015), where the author discusses "the unreasonable effectiveness of recurrent neural networks", he reflects: "I'm training RNN's all the time, and I've witnessed their power and robustness many times, and yet their magical outputs still find ways of amusing me."

A very simple RNN

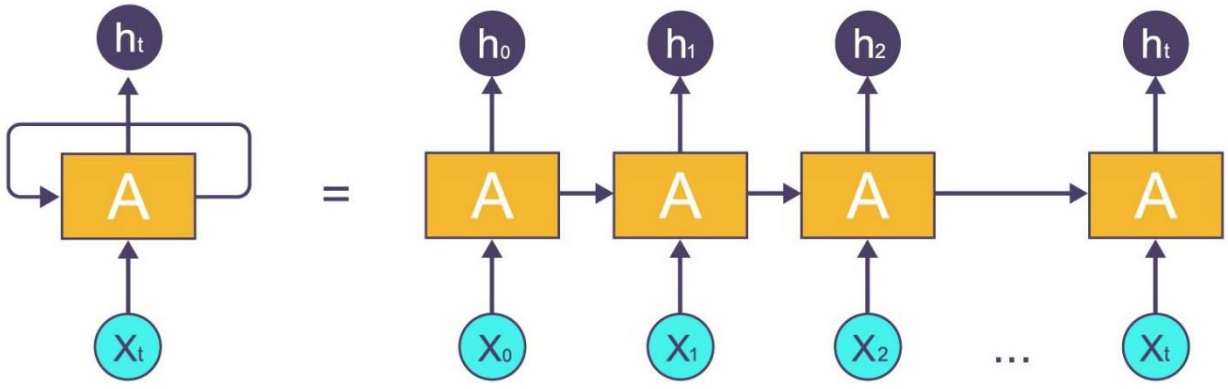


Figure 19: The simplest possible RNN (Adapted from Olah, 2015)

The figure above shows the simplest possible RNN. At each time step t, the neuron receives two inputs: the observation vector $\mathbf{x}_t$, and its state from the previous time-step, $\mathbf{h}_{t-1}$. The (hidden) state $\mathbf{h}_t$ is output at each step. (Note that the output $\mathbf{h}_t$ is available for further processing if required.) In the right-most image, the RNN is unrolled over time.

The actual computation for this simple RNN should look familiar:

$$h_t = \Phi(W_x x_t + W_h h_t + b),$$

where Φ is the activation function. Each recurrent neuron has two sets of weights shared over all time steps, W_x and W_h. If the dimension of the input vector $\mathbf{x}$ is d, and that of the output vector $\mathbf{h}$ is n, then W_h is an $n \times n$, and W_x a $d \times n$ matrix. The activation function Φ is applied pointwise as before.

We can combine the two weight matrices into one and write:

$$h_t = \Phi(W[x_t\ ,\ h_{t-1}] + b).$$

Note that:

- Since $\mathbf{h}_t$ depends on $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$, which depends on $\mathbf{h}_{t-2}$ and $\mathbf{x}_{t-1}$, and so on, $\mathbf{h}_t$ depends on all the time inputs right up to the beginning.
- A part of a network that preserves some part of its state over time-steps is called a **memory cell** or just a **cell**.
- In general, the (hidden) state of a cell $\mathbf{h}_t$ is a function of its state at the previous time step $\mathbf{h}_{t-1}$, and an input at the current time step $\mathbf{x}_t$ – i.e., $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$.

The different structures of an RNN

The next figure is taken from Karpathy and illustrates the different forms that RNN's can acquire.

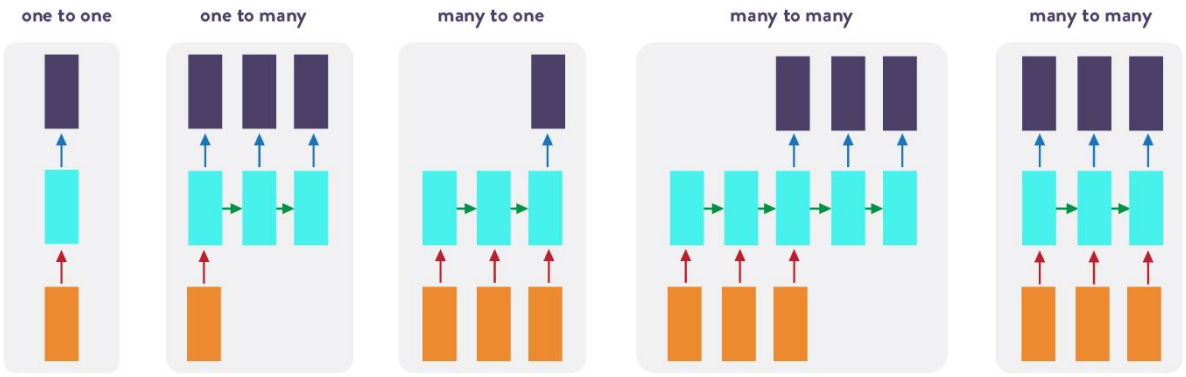


Figure 20: The different structures of an RNN (Adapted from Karpathy, 2015)

Examples

To start with, we have a **one-to-one network**. It is included here for the sake of covering all of the structures but, for this, you may as well use a feedforward network.

A **one-to-many network** may be an example of music generation, where we provide a single note as an input and the model generates the rest of the song as multiple outputs.

A **many-to-one network** may be used in predicting the direction of a next-day move in alpha design strategies. Here, the model is fed a sequence of time series data and outputs the next day's direction.

Lastly, we get to the **many-to-many network**. This is generally split into two. The first may be a sequence-to-sequence model, such as machine translation – say, English to French. The other may be a multi-class labeling model where we provide a video to the model and we require it to label each frame.

A basic RNN in TensorFlow

Here we essentially follow the open source notebook of Aurélien Geron, (2018) Chapter 14, and write a basic RNN in TensorFlow. Note how easy it is if we make use of the powerful functions in TensorFlow.

In [1]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm

%matplotlib inline

# To plot pretty figures
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
```

Save and output settings

Below, we set up some global variables that will allow the output to be stable across runs, as well as where to save images.

In [2]:
```
# to make this notebook's output stable across runs
def reset_graph(seed=42):
    tf.reset_default_graph()
    tf.set_random_seed(seed)
    np.random.seed(seed)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "rnn"

def save_fig(fig_id, tight_layout=True):
    path = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID, fig_id + ".png")
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format='png', dpi=300)
```

Create a time series

Next, we generate a time series using the following formula:

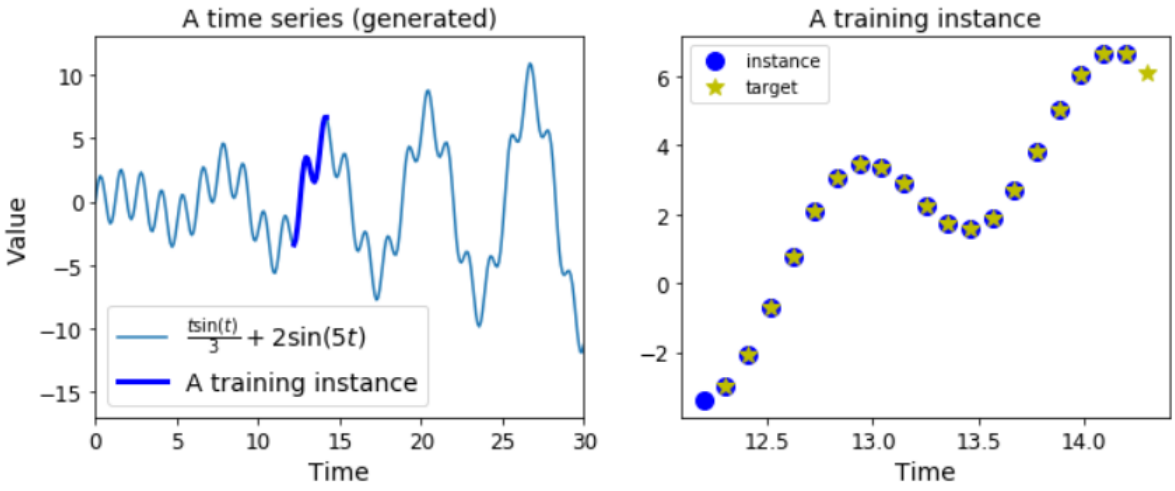
$$\frac{t\ sin(t)}{3} + 2\ sin(5t).$$

Functions

In [3]:
```
t_min, t_max = 0, 30
resolution = 0.1

def time_series(t):
    # Create a time series.
    return t * np.sin(t) / 3 + 2 * np.sin(t*5)

def next_batch(batch_size, n_steps):
    """
    Generate consecutive samples of time series from the time series.
    The starting position of the samples is arbitrary.

    Parameters
    ----------
    batch_size: int
                The number of series in the batch
    n_steps: int
            The number of samples from the time series in each batch

    Return
    ------
    X_batch: (batch_size, n_steps, 1) array
             batch_size samples each of length n_steps, and each sample is 1d
    y_batch: (batch_size, n_steps, 1) array
             targets, shifted one position from X_batch
    """
    t0 = np.random.rand(batch_size, 1) * (t_max - t_min - n_steps * resolution)
    Ts = t0 + np.arange(0., n_steps + 1) * resolution
    ys = time_series(Ts)
    return ys[:, :-1].reshape(-1, n_steps, 1), ys[:, 1:].reshape(-1, n_steps, 1)
```

Create series

In [4]:
```
t = np.linspace(t_min, t_max, int((t_max - t_min) / resolution))

n_steps = 20
t_instance = np.linspace(12.2, 12.2 + resolution * (n_steps + 1), n_steps + 1)

# Plot full series
plt.figure(figsize=(11,4))
plt.subplot(121)
plt.title("A time series (generated)", fontsize=14)
plt.plot(t, time_series(t), label=r"$\frac{t\sin(t)}{3} + 2 \sin(5t)$")
plt.plot(t_instance[:-1], time_series(t_instance[:-1]), "b-", linewidth=3, label="A training instance")
plt.legend(loc="lower left", fontsize=14)
plt.axis([0, 30, -17, 13])
plt.xlabel("Time")
plt.ylabel("Value")

# Plot training instance
plt.subplot(122)
plt.title("A training instance", fontsize=14)
plt.plot(t_instance[:-1], time_series(t_instance[:-1]), "bo", markersize=10, label="instance")
plt.plot(t_instance[1:], time_series(t_instance[1:]), "y*", markersize=10, label="target")
plt.legend(loc="upper left")
plt.xlabel("Time")

plt.show()
```



```
In [5]: # Generate a single training instance
        X_batch, y_batch = next_batch(1, n_steps)
```

```
In [6]: # Each value in the sequence predicts the next value in the sequence.
        np.c_[X_batch[0], y_batch[0]]
```

```
Out[6]: array([[ 1.15352294,  2.00049277],
               [ 2.00049277,  2.46949091],
               [ 2.46949091,  2.45385795],
               [ 2.45385795,  1.96445306],
               [ 1.96445306,  1.12676875],
               [ 1.12676875,  0.14999406],
               [ 0.14999406, -0.724396  ],
               [-0.724396  , -1.28193394],
               [-1.28193394, -1.38781614],
               [-1.38781614, -1.02002678],
               [-1.02002678, -0.27481718],
               [-0.27481718,  0.65680343],
               [ 0.65680343,  1.5358113 ],
               [ 1.5358113 ,  2.13370285],
               [ 2.13370285,  2.28849002],
               [ 2.28849002,  1.94444692],
               [ 1.94444692,  1.16587652],
               [ 1.16587652,  0.12152578],
               [ 0.12152578, -0.95653592],
               [-0.95653592, -1.82950897]])
```

Basic RNN

Now that we have generated the series, let's continue to build a model that will be useful in predicting the series.

Construction

We are all familiar with the construction and execution steps. For the basic RNN, we use `tf.contrib.rnn.BasicRNNCell` to create the cells, with 100 neurons, using ReLU activation functions. We are doing a regression style problem and for this we will make use of the MSE and ADAM to optimize.

In [7]:
```
reset_graph()

n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

learning_rate = 0.001

with tf.name_scope("Inputs") :
    X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
    y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])

with tf.name_scope("Cell"):
    cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
    rnn_outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)

with tf.name_scope("Outputs"):
    stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])
    stacked_outputs = tf.layers.dense(stacked_rnn_outputs, n_outputs)
    outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])

with tf.name_scope("Training"):
    loss = tf.reduce_mean(tf.square(outputs - y))
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope("Init_and_Saver"):
    init = tf.global_variables_initializer()
    saver = tf.train.Saver()
```

Execution

Run for 1,500 iterations with a batch size of 50.

In [8]:
```
n_iterations = 1500
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

    # Generate a test sequence
    X_new = time_series(np.array(t_instance[:-1].reshape(-1, n_steps, n_inputs)))

    # Predict on the test sequence
    y_pred = sess.run(outputs, feed_dict={X: X_new})
    saver.save(sess, "./my_time_series_model")
```

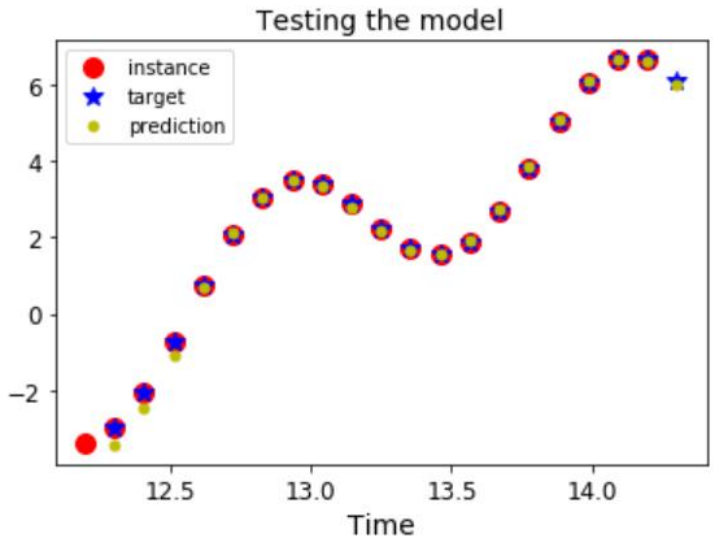
```
0       MSE: 13.907029
100     MSE: 0.5056698
200     MSE: 0.19735886
300     MSE: 0.101214476
400     MSE: 0.06850145
500     MSE: 0.06291986
600     MSE: 0.055129297
700     MSE: 0.049436502
800     MSE: 0.050434686
900     MSE: 0.0482007
1000    MSE: 0.04809868
1100    MSE: 0.04982501
1200    MSE: 0.041912545
1300    MSE: 0.049292978
1400    MSE: 0.043140374
```

In [9]:
```
# The following are the predictions on the test sequence
y_pred
```

Out[9]:
```
array([[[-3.4332483],
        [-2.4594698],
        [-1.1081185],
        [ 0.6882153],
        [ 2.1105688],
        [ 3.0585155],
        [ 3.5144088],
        [ 3.3531117],
        [ 2.808016 ],
        [ 2.1606152],
        [ 1.662645 ],
        [ 1.5578941],
        [ 1.9173537],
        [ 2.7210245],
        [ 3.8667865],
        [ 5.100083 ],
        [ 6.099999 ],
        [ 6.6480975],
        [ 6.6147423],
        [ 6.022089 ]]], dtype=float32)
```

Plot actual vs prediction

In [10]:
```
plt.title("Testing the model", fontsize=14)
plt.plot(t_instance[:-1], time_series(t_instance[:-1]), "ro", markersize=10, label="instance")
plt.plot(t_instance[1:], time_series(t_instance[1:]), "b*", markersize=10, label="target")
plt.plot(t_instance[1:], y_pred[0,:,0], "y.", markersize=10, label="prediction")
plt.legend(loc="upper left")
plt.xlabel("Time")

plt.show()
```



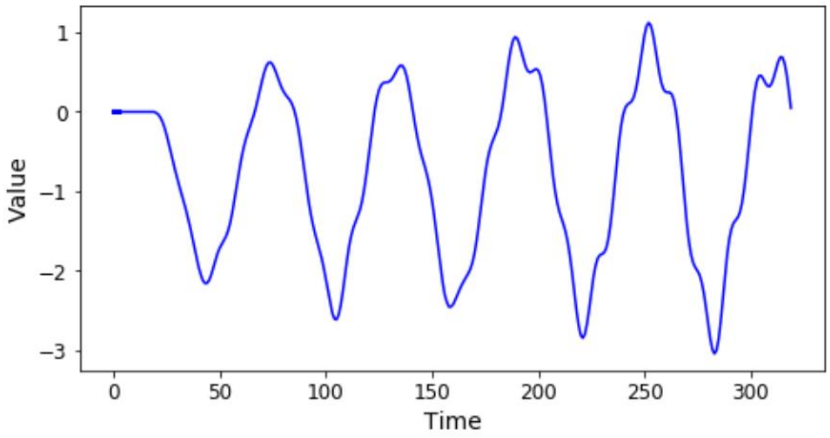
We can see that the basic RNN has learnt the function and has a good fit.

Generate output

In [11]:
```
with tf.Session() as sess:
    saver.restore(sess, "./my_time_series_model")
    sequence = [0.] * n_steps
    for iteration in range(300):
        X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_batch})
        sequence.append(y_pred[0, -1, 0])
```

```
INFO:tensorflow:Restoring parameters from ./my_time_series_model
```

In [12]:
```
plt.figure(figsize=(8,4))
plt.plot(np.arange(len(sequence)), sequence, "b-")
plt.plot(t[:n_steps], sequence[:n_steps], "b-", linewidth=3)
plt.xlabel("Time")
plt.ylabel("Value")
plt.show()
```



Unfortunately, the basic RNN has not learnt that the output is increasing in magnitude as time goes on.

Long short-term memory cells

There are two basic problems associated with the basic cells discussed above, and both were carefully avoided in the examples.

The first is that, in order to train the RNN on long sequences, it becomes (for training purposes) like a very deep neural network, since the gradient of the cost function is backpropagated through the whole network. It therefore suffers from the same vanishing/exploding gradient problem of deep neural networks. Of course, some of the same tricks used for deep neural networks can still be useful.

The second problem is that part of the memory of the first inputs gradually fades away as more time-steps are traversed. If the value to be predicted only relies on the immediate past, this is not too much of a problem. In many cases, however, the context needed for an accurate prediction lies far in the past. For example, in the sentence “There is a cloud in the ...”, the immediate context makes it quite likely that the missing word is “sky”. On the other hand, let's consider the statement, “She lives in Oaxaca. She speaks ...”. The immediate context indicates that the missing word is a language. Which language, however, cannot be derived from the immediate past. That context is provided by the earlier statement.

For this, we need more complex cells, and one of the most popular is the **long short-term memory** (LTSM) cell of Hochreiter and Schmidhuber (1997).

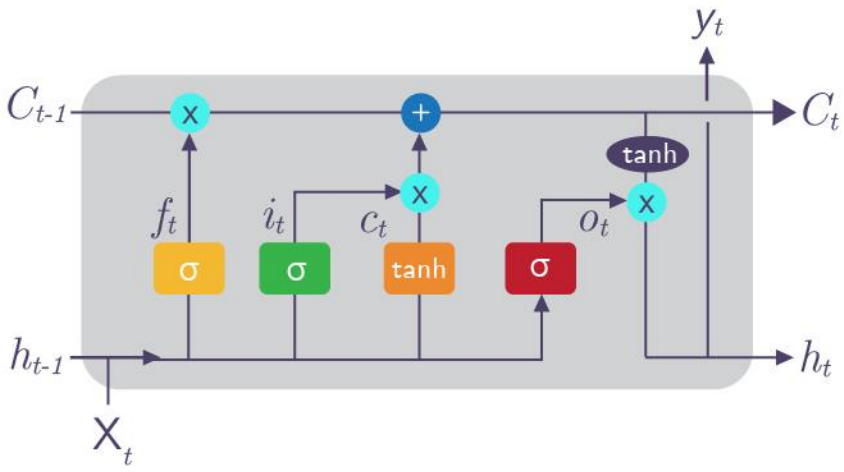


Figure 21: The LSTM cell (Adapted from Olah, 2015)

In the figure above, the LSTM cell maintains two states, $\mathbf{c}_t$ and $\mathbf{h}_t$. The state at the top $\mathbf{c}_t$ is the cell state, and Olah calls it a kind of "conveyor belt". It carries information straight through the entire chain with only minor changes. Memory is removed by the **forget gate** and new memory is added by the **input gate** and the result is passed on. The gates determine which part of the information should pass through, and which part should be removed. The gates consist of a sigmoid neural layer, and a pointwise multiplication. Since the sigmoid layer outputs values between 0 and 1, the cell vector is multiplied by values between 0 and 1.

A value of 1 ensures that that particular component of the state is perfectly preserved, while a value of zero completely removes that component from memory.

In each LSTM cell, there are three gates – a forget gate, an input gate, and an output gate. Let us look at these in more detail.

1 The **forget gate** takes as input the short-term memory $\mathbf{h}_{t-1}$ and the input $\mathbf{x}_t$, and outputs

$$\mathbf{f}_t = \sigma(W_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f).$$

This gate controls how much memory is destroyed.

2 The **input gate** also takes as input the short-term memory $\mathbf{h}_{t-1}$ and the input $\mathbf{x}_t$, and outputs,

$$\mathbf{i}_t = \sigma(W_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i),$$

$$\tilde{\mathbf{c}}_t = tanh(W_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c).$$

This gate controls how much memory is added at this time.

The long-term memory is now updated by deleting part of its memory using the forget gate, and adding to its memory using the input gate,

$$\mathbf{c}_t = \mathbf{f}_t * \mathbf{c}_{t-1} + \mathbf{i}_t * \tilde{\mathbf{c}}_t.$$

3 Finally, the output gate updates the short-term memory, which is also the cell output. The input gate also takes as input the short-term memory $\mathbf{h}_{t-1}$, the input $\mathbf{x}_t$ and outputs,

$$\mathbf{o}_t = \sigma(W_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o),$$

$$\mathbf{h}_t = \mathbf{o}_t * tanh(\mathbf{c}_t).$$

Construction

Now we construct a LSTM model using `tf.contrib.rnn.BasicLSTMCell` to create the cells, with 100 neurons, using ReLU activation functions. We are doing a regression style problem and for this we will make use of the MSE and ADAM to optimize, just as before.

In [13]:
```
reset_graph()

n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

learning_rate = 0.001

with tf.name_scope("Inputs") :
    X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
    y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])

with tf.name_scope("Cell"):
    lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons, activation=tf.nn.relu)
    lstm_outputs, states = tf.nn.dynamic_rnn(lstm_cell, X, dtype=tf.float32)

with tf.name_scope("Outputs"):
    stacked_lstm_outputs = tf.reshape(lstm_outputs, [-1, n_neurons])
    stacked_outputs = tf.layers.dense(stacked_lstm_outputs, n_outputs)
    outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])

with tf.name_scope("Training"):
    loss = tf.reduce_mean(tf.square(outputs - y))
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope("Init_and_Saver"):
    init = tf.global_variables_initializer()
    saver = tf.train.Saver()
```

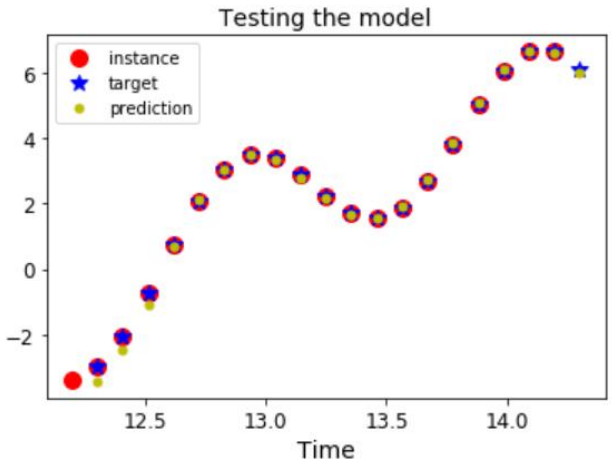
Execution

Run for 1,500 iterations, batch size 50.

In [14]:
```
n_iterations = 1500
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

    # Generate a test sequence
    X_new = time_series(np.array(t_instance[:-1].reshape(-1, n_steps, n_inputs)))

    # Predict on the test sequence
    y_pred = sess.run(outputs, feed_dict={X: X_new})
    saver.save(sess, "./my_lstm_time_series_model")
```

```
0       MSE: 15.115627
100     MSE: 1.4786355
200     MSE: 0.5006361
300     MSE: 0.30985782
400     MSE: 0.118771255
500     MSE: 0.06286281
600     MSE: 0.04371503
700     MSE: 0.04437162
800     MSE: 0.043510098
900     MSE: 0.052118573
1000    MSE: 0.049101226
1100    MSE: 0.038297866
1200    MSE: 0.034967538
1300    MSE: 0.0431967
1400    MSE: 0.037618563
```

In [15]:
```
plt.title("Testing the model", fontsize=14)
plt.plot(t_instance[:-1], time_series(t_instance[:-1]), "ro", markersize=10, label="instance")
plt.plot(t_instance[1:], time_series(t_instance[1:]), "b*", markersize=10, label="target")
plt.plot(t_instance[1:], y_pred[0,:,0], "y.", markersize=10, label="prediction")
plt.legend(loc="upper left")
plt.xlabel("Time")

plt.show()
```



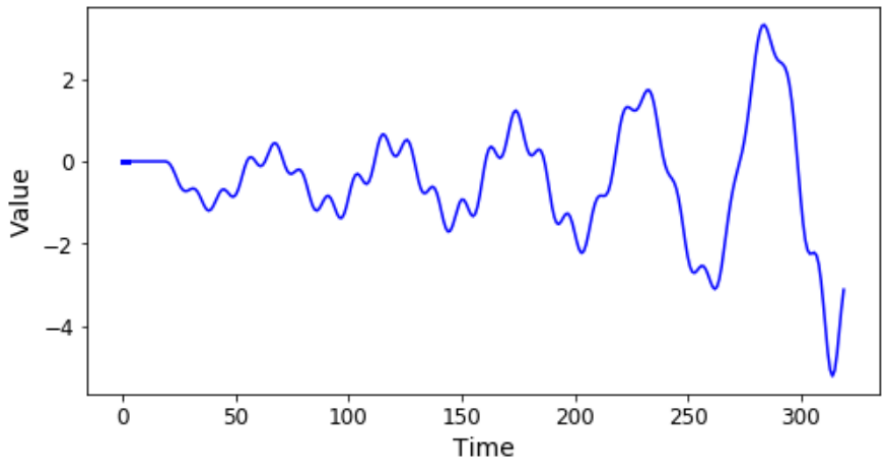
We can see that the LSTM has learnt the function and has a good fit.

In [16]:
```
with tf.Session() as sess:
    saver.restore(sess, "./my_lstm_time_series_model")
    sequence = [0.] * n_steps
    for iteration in range(300):
        X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_batch})
        sequence.append(y_pred[0, -1, 0])
```

```
INFO:tensorflow:Restoring parameters from ./my_lstm_time_series_model
```

Generate output

In [17]:
```
plt.figure(figsize=(8,4))
plt.plot(np.arange(len(sequence)), sequence, "b-")
plt.plot(t[:n_steps], sequence[:n_steps], "b-", linewidth=3)
plt.xlabel("Time")
plt.ylabel("Value")
plt.show()
```



The LSTM cell has done better than the basic RNN as it has learnt that, as time progresses, the magnitude increases.

The gated recurrent unit

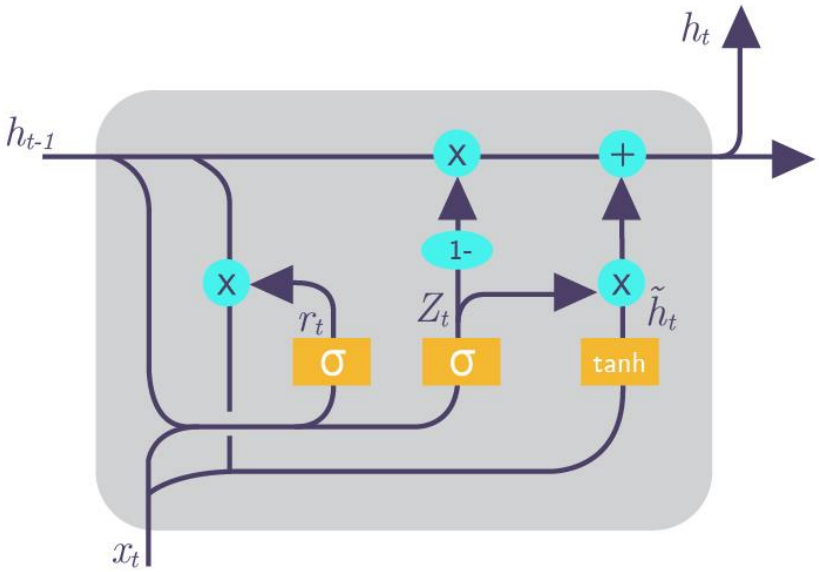


Figure 22: GRU cell (Adapted from Olah, 2015)

There are a number of variations of the LSTM cell, and one of the most important is the GRU cell. It is a simplification of the LSTM cell, and performs just as well, which explains its popularity. Its main features, as compared with the LSTM cell, are:

- There is a single state vector $\mathbf{h}_t$.
- A single controller controls both the forget gate and the input gate. If the controller outputs a 1, the input gate is open and the forget gate is closed. On the other hand, if it outputs a 0, the input gate is closed and the output gate is open. This means that if a memory is to be stored, it has to be erased first.
- There is no output gate as the full state vector is output at every step. There is a new controller that determines which part of the previous state is exposed to the main layer.

Looking at the mathematics will make this clear:

$$\mathbf{z}_t = \sigma(W_z[\mathbf{h}_{t-1}, \mathbf{x}_t]),$$

$$\mathbf{r}_t = \sigma(W_r[\mathbf{h}_{t-1}, \mathbf{x}_t]),$$

$$\mathbf{h}_t = \tanh(W[\mathbf{r}_t * \mathbf{h}_{t-1}, \mathbf{x}_t]),$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) * \mathbf{h}_{t-1} + \mathbf{z}_t * \mathbf{h}_t .$$

Construction

Now we construct a GRU model using `tf.contrib.rnn.GRUCell` to create the cells, with 100 neurons, using ReLU activation functions. We are doing a regression style problem and for this we will make use of the MSE and ADAM to optimize, just as before.

In [13]:
```
reset_graph()

n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

learning_rate = 0.001

with tf.name_scope("Inputs") :
    X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
    y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])

with tf.name_scope("Cell"):
    lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons, activation=tf.nn.relu)
    lstm_outputs, states = tf.nn.dynamic_rnn(lstm_cell, X, dtype=tf.float32)

with tf.name_scope("Outputs"):
    stacked_lstm_outputs = tf.reshape(lstm_outputs, [-1, n_neurons])
    stacked_outputs = tf.layers.dense(stacked_lstm_outputs, n_outputs)
    outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])

with tf.name_scope("Training"):
    loss = tf.reduce_mean(tf.square(outputs - y))
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope("Init_and_Saver"):
    init = tf.global_variables_initializer()
    saver = tf.train.Saver()
```

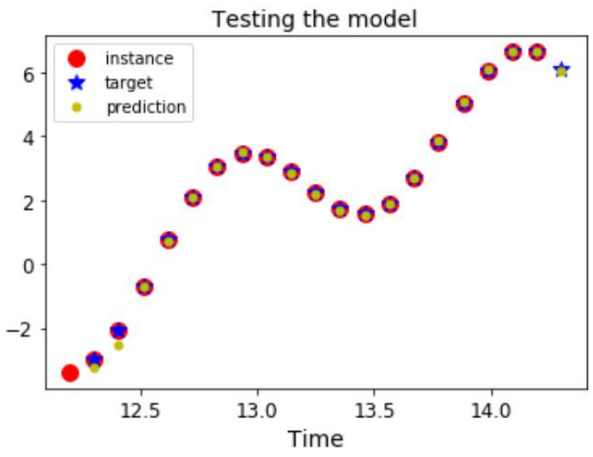
Execution

Run for 1,500 iterations, batch size 50.

In [32]:
```
n_iterations = 1600
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

    # Generate a test sequence
    X_new = time_series(np.array(t_instance[:-1].reshape(-1, n_steps, n_inputs)))

    # Predict on the test sequence
    y_pred = sess.run(outputs, feed_dict={X: X_new})
    saver.save(sess, "./my_gru_time_series_model")
```

```
0       MSE: 18.001982
100     MSE: 1.236871
200     MSE: 0.56422
300     MSE: 0.38811332
400     MSE: 0.20370233
500     MSE: 0.10633521
600     MSE: 0.059438452
700     MSE: 0.053997226
800     MSE: 0.04914086
900     MSE: 0.059127163
1000    MSE: 0.05402688
1100    MSE: 0.040683806
1200    MSE: 0.037607495
1300    MSE: 0.047508392
1400    MSE: 0.04392034
1500    MSE: 0.053137667
```

In [33]:
```
plt.title("Testing the model", fontsize=14)
plt.plot(t_instance[:-1], time_series(t_instance[:-1]), "ro", markersize=10, label="instance")
plt.plot(t_instance[1:], time_series(t_instance[1:]), "b*", markersize=10, label="target")
plt.plot(t_instance[1:], y_pred[0,:,0], "y.", markersize=10, label="prediction")
plt.legend(loc="upper left")
plt.xlabel("Time")

plt.show()
```



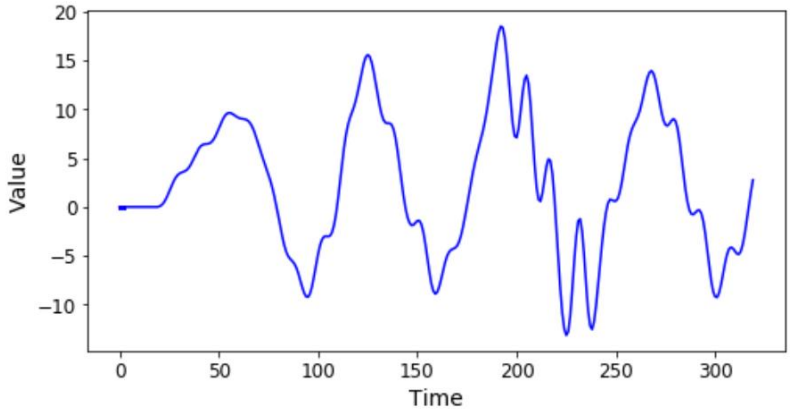
Again, we can see that the GRU has a good fit.

In [34]:
```
with tf.Session() as sess:
    saver.restore(sess, "./my_gru_time_series_model")
    sequence = [0.] * n_steps
    for iteration in range(300):
        X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_batch})
        sequence.append(y_pred[0, -1, 0])
```

```
INFO:tensorflow:Restoring parameters from ./my_gru_time_series_model
```

Generate output

In [35]:
```
plt.figure(figsize=(8,4))
plt.plot(np.arange(len(sequence)), sequence, "b-")
plt.plot(t[:n_steps], sequence[:n_steps], "b-", linewidth=3)
plt.xlabel("Time")
plt.ylabel("Value")
plt.show()
```



Here we see something interesting in that the output generated is not what we would expect. However, the model is fitting the data rather well.

Additional research

There is an interesting presentation by Dr Matthew Dixon. In his lecture titled "The Neural Networks Survival Kit for Quants" he attempts to debunk some of the common myths about Neural Networks. Slides to his presentation are found here.

This week's academic paper in review is also by the same author, entitled "Sequence Classification of the Limit Order Book using Recurrent Neural Networks". (Please note that some of the Module 6 multiple-choice questions may be based on this.)

Bibliography

Dixon, M. (2017). "Sequence Classification of the Limit Order Book using Recurrent Neural Networks. *Journal of Computational Science*, 24:277-286.

Fehlhaber, K. (2014). "Hubel and Wiesel & the Neural Basis of Visual Perception." [Online] Available at: https://knowingneurons.com/2014/10/29/hubel-and-wiesel-the-neural-basis-of-visual-perception/

Hochreiter, S. & Schmidhuber, J. (1997). "Long Short-term Memory". *Neural Computation*, 9(8):1735-1780.

Karpathy, A. (2015). "The Unreasonable Effectiveness of Recurrent Neural Networks." [Online] Available at: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Olah, C. (2015). "Understanding LTMS Networks." [Online] Available at: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.