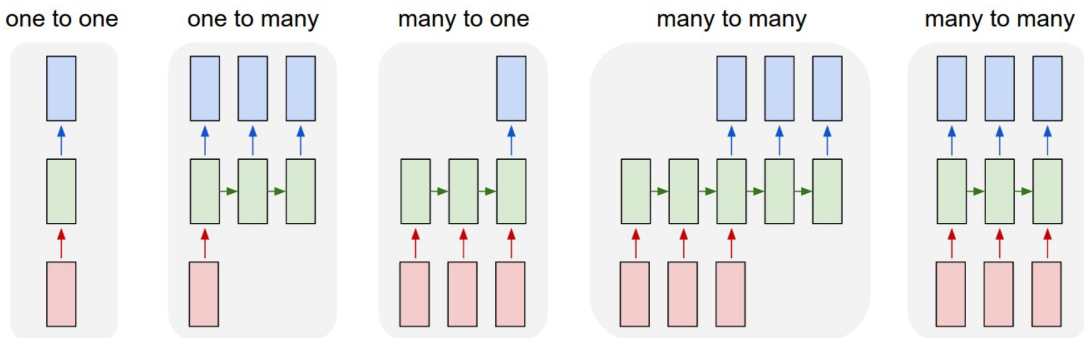
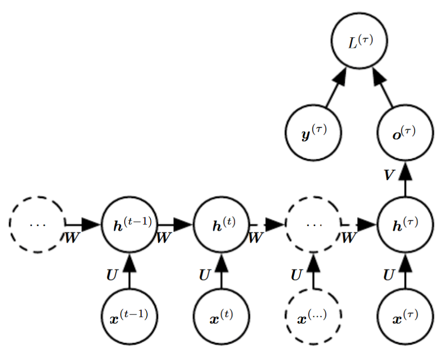
14 Recurrent neural networks

Recurrent neural networks (RNNs) are designed for features that are recurrent (e.g., time series). A good description of the various applications of RNNs is in Karpathy's blog. This figure from that blog illustrates various purposes of RNNs.



14.1 Many-to-one RNN models

The "**many-to-one**" type is probably the most common type of RNN models. For example, we may want to use the weather information in the past 10 days to predict the temperature tomorrow. It can also be illustrated below (DL Figure 10.5), where $o^{(r)}$ is the predicted value.



For this model, each observation consists of a set of input vectors, $\{x^{(1)}, \ldots, x^{(k)}\}$, and an output y, which can be a scalar or a vector. The value k may vary from one observation to another. The model is effectively a feedforward neural network with k hidden layers: $h^{(t)}$ $(t = 1, \cdots, k)$; we set $h^{(0)} = 0$. All these layers have the same number of nodes (often called **units** in RNN). Each hidden layer $h^{(t)}$ has inputs not only from the previous layer $h^{(t-1)}$, but also from the input vector $x^{(t)}$. The output layer has inputs from $h^{(k)}$. A key feature of the RNN is that all these hidden layers **share the same weights** W and U and **bias** b:

$$h^{(t)} = f_R(Wh^{(t-1)} + Ux^{(t)} + b) \quad (t = 1, \cdots, k),$$

where $f_R()$ is an activation function (tanh() is often used). After we have gone through all the input vectors, the output layer has weights V and c:

$$\hat{y} = f_y(Vh^{(k)} + c),$$

where $f_y()$ is an activation function. If the input vector $x^{(t)}$ is p-dimensional and each RNN layer $h^{(t)}$ has q units, then W is $q \times q$, U is $q \times p$, and b is $q \times 1$. So the total number of parameters for the RNN layer is $q(p + q + 1)$.

In summary, when this RNN model is applied to an observation with k input vectors, it works in the following way:

- Initialize vector $h^{(0)} = 0$;
- At step t $(t = 1, \ldots, k)$, $h^{(t)} = f_R(Wh^{(t-1)} + Ux^{(t)} + b)$;
- $\hat{y} = f_y(Vh^{(k)} + c)$.

Interpretation: This RNN model can be viewed from the following perspective. There is a hidden vector h that carries intermediate information between the input vectors and the output. It is updated repeatedly at every new input vector, taking into account its previous value and the new input vector. At the end, the hidden vector is translated to an outcome.

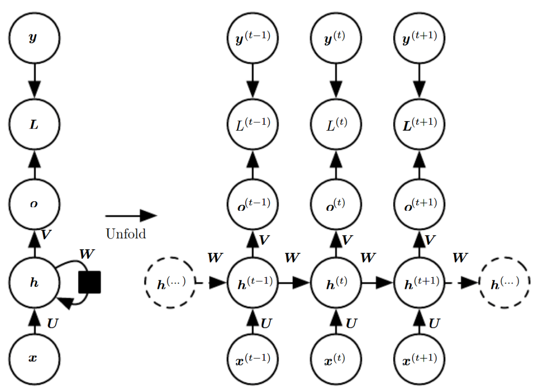
Calculation of the gradients: Let $L(y, \hat{y})$ be the cost function, where the predicted value $\hat{y}$ is a function of $(x^{(1)}, \ldots, x^{(k)})$. We first treat the individual W's in the k steps as if they are different parameters, and calculate their corresponding gradients. Then the gradient of $L(y, \hat{y})$ with respect to W is the sum of the gradients with respect to all the individual W's. Similarly for the gradients with respect to U and b.

Below is an example Keras code for such a "many-to-one" RNN model. In this example, $p = 10$ and $q = 32$; the number of parameters for the RNN layer is $32 \times (10 + 32 + 1) = 1376$. The output is a scalar (i.e., 1 unit), and it requires $32 + 1 = 33$ parameters. The default activation function for the RNN layer is `activation = "tanh"`.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_simple_rnn(units = 32, input_shape = list(NULL, 10)) %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

14.2 Many-to-many RNN models

The "**many-to-many**" type can also be illustrated below (DL Figure 10.3). The model is often shown with a loop as the one on the left; its unfolded version is on the right.



For this model, each observation consists of a set of k pairs of input vectors and outputs, $\{x^{(1)}, \ldots, x^{(k)}, y^{(1)}, \ldots, y^{(k)}\}$, where k may vary from one observation to another. For example, in word processing, an observation can be a sentence, with $x^{(t)}$ being a "phrase" of 5 contiguous words and $y^{(t)}$ being the next word. Then a sentence with 15 words would have $k = 10$ pairs of $(x^{(t)}, y^{(t)})$.

When this RNN model is applied to an observation of size k, it works in the following way:

- Initialize vector $h^{(0)} = 0$.
- At step t ($t = 1, \ldots, k$), there are
  - $h^{(t)} = f_R(Wh^{(t-1)} + Ux^{(t)} + b)$;
  - $\hat{y}^{(t)} = f_y(Vh^{(t)} + c)$.

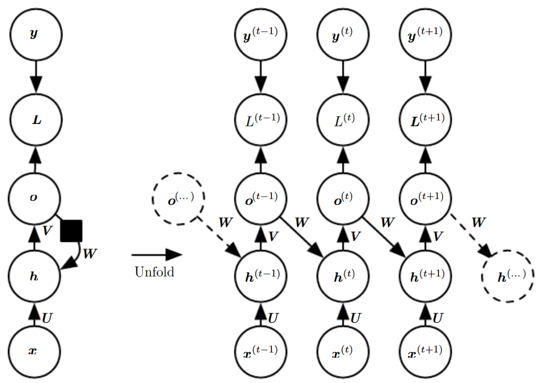
Similar to the "many-to-one" model, this model has the **shared weights** W and U and **bias** b. Each hidden layer $h^{(t)}$ has inputs from the previous layer $h^{(t-1)}$ and the input vector $x^{(t)}$. Each output $\hat{y}^{(t)}$ has inputs from $h^{(t)}$. The paths to $\hat{y}^{(t)}$ ($t = 1, \ldots, k$) also **share the same weights** V and **bias** c. The "many-to-one" and "many-to-many" models are very similar; the only difference is that the former has the output at the end while the latter outputs an outcome at every step. Both models have the same number of parameters, $q(p + q + 1)$.

The cost for an observation of size k is $\sum_{t=1}^{k} L(y^{(t)}, \hat{y}^{(t)})$, where $\hat{y}^{(t)}$ is a function of the first t input vectors $(x^{(1)}, \ldots, x^{(t)})$. Similar to the "many-to-one" model, because of parameter sharing, the gradient of $L(y, \hat{y})$ with respect to W is the sum of the gradients with respect to all the individual W's. Similarly for the gradients with respect to all other parameters U, b, V, c.

Below is an example Keras code for such a "many-to-many" RNN layer. Here, `return_sequences=T` tells the model to output $h^{(t)}$ to the next Keras layer at every t, and `time_distributed()` tells the model to compute and evaluate the outcome at every t. Again, the default activation function for the RNN layer is `activation = "tanh"`.

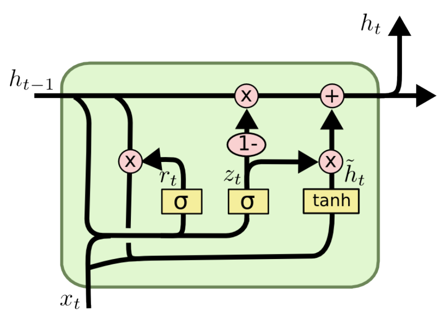
```
library(keras)
model <- keras_model_sequential() %>%
  layer_simple_rnn(units = 32, return_sequences=T, input_shape = list(NULL, 10)) %>%
  time_distributed(layer_dense(units = 1, activation = "sigmoid"))

summary(model)
```

An variation of this "many-to-many" model is below (DL Figure 10.4), which differs from the model above in that $h^{(t)} = \tanh(Wo^{(t-1)} + Ux^{(t)} + b)$, where $o^{(t)} = \hat{y}^{(t)} = Vh^{(t)} + c$. Here the dimensions of W are $q \times a$, where a is the dimension of $o^{(t)}$.



14.3 GRUs and LSTMs

In the above sections, $h^{(t)}$ depends on $h^{(t-1)}$ and $x^{(t)}$ in a relatively simple way, a linear combination plus a bias and then followed by an activation function: $h^{(t)} = f_R(Wh^{(t-1)} + Ux^{(t)} + b)$. In some RNNs, $h^{(t)}$ depends on $h^{(t-1)}$ and $x^{(t)}$ in a sophisticated fashion. One example is the **gated recurrent unit** (GRU) (Cho et al., 2014). The figure below is a "fully gated" version of GRU (from Olah's blog).



Below is the matrix representation of the operations for a layer with q GRUs:

$$\begin{cases} z_t &= \sigma(W_z x_t + R_z h_{t-1} + b_z), \qquad \text{(update gate)} \\ r_t &= \sigma(W_r x_t + R_r h_{t-1} + b_r), \\ \tilde{h}_t &= f(W_a x_t + R_a(r_t \circ h_{t-1}) + b_a), \\ h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t. \end{cases}$$

Here, z_t, r_t, $\tilde{h}_t$, h_t are vectors with the same length q, and the operator $\circ$ is the Hadamard product (i.e., element-wise product between vectors). For a single unit (i.e., $q = 1$), $\circ$ becomes the regular product. The activation function $f()$

is often tanh() in practice.

Interpretation: In every GRU, $\tilde{h}_t$ is a new candidate h value, and z_t is the "update gate". If z_t could take value 0 or 1, it would determine whether h_t is updated or not. For example, if $z_t = 1$, h_t is updated to be $\tilde{h}_t$; if $z_t = 0$, h_t keeps the previous value h_{t-1}. But since $0 < z_t < 1$, z_t serves as a weight for $\tilde{h}_t$, and $1 - z_t$ is the weight for h_{t-1}.

If the input $x^{(t)}$ is p-dimensional and a GRU layer has q GRUs, all W matrices are $q \times p$, all R matrices are $q \times q$, and all b vectors are $q \times 1$. So the total number of parameters for the GRU layer is $3q(p + q + 1)$.

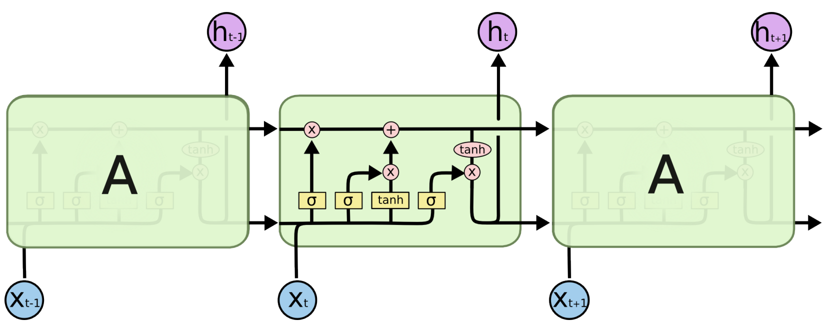
Below is an example Keras code for a layer of 32 GRUs. The default activation function for the GRU layer is `activation = "tanh"`. In this example, the number of parameters for the GRU layer is $3 \times 32 \times (10+32+1) = 4128$. The 1-dimensional output requires $32 + 1 = 33$ parameters.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_gru(units = 32, input_shape = list(NULL, 10)) %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

Stacking GRU layers: One can stack a GRU layer after another by specifying `return_sequences=T` in the first GRU layer so that it feeds its $h^{(t)}$ as the input vector to the next GRU layer. The next GRU layer treats the $h^{(t)}$ from the previous GRU layer as its $x^{(t)}$. In the example below, the second GRU layer has 64 units, requiring $3 \times 64 \times (32 + 64 + 1) = 18,624$ parameters.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_gru(units = 32, return_sequences=T, input_shape = list(NULL, 10)) %>%
  layer_gru(units = 64, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

LSTMs: GRUs are a simplified version of **long short-term memory** (LSTM) networks (Hochreiter and Schmidhuber, 1997). An LSTM unit looks like the figure below. It has a separate "memory cell" c_t in addition to the "hidden" node h_t. A nice description of the LSTM is given by Christopher Olah.



Below is the matrix representation of the operations for a layer with q LSTM units:

$$\begin{cases} a_t &= \tanh(W_a x_t + R_a h_{t-1} + b_a), & \text{input activation} \\ f_t &= \sigma(W_f x_t + R_f h_{t-1} + b_f), & \text{forget gate} \\ i_t &= \sigma(W_i x_t + R_i h_{t-1} + b_i), & \text{input gate} \\ o_t &= \sigma(W_o x_t + R_o h_{t-1} + b_o), & \text{output gate} \\ c_t &= f_t \circ c_{t-1} + i_t \circ a_t, & \text{memory cell state} \\ h_t &= o_t \circ \tanh(c_t), & \text{output} \end{cases}$$

Here, the "forget gate" f_t determines how much of the previous memory c_{t-1} is retained, and the "input gate" i_t determines how much of the new "input activation" a_t will be used towards the new memory c_t. A version of LSTM has $c_t = f_t \circ c_{t-1} + (1 - f_t) \circ a_t$ instead, combining the forget and input gates into one.

An LSTM layer with q units and p-dimensional input has $4q(p+q+1)$ parameters. Below is an example Keras code for a layer of 32 GRUs. The default activation function for the GRU layer is `activation = "tanh"`. In this example, the number of parameters for the LSTM layer is $4 \times 32 \times (10+32+1) = 5504$.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_lstm(units = 32, input_shape = list(NULL, 10)) %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

14.4 Bidirectional RNNs

A bidirectional RNN with q units in each direction has two parallel RNN layers, one operating forwardly from $x^{(1)}$ to $x^{(k)}$ and the other operating backwardly from $x^{(k)}$ to $x^{(1)}$. At the end, the two final hidden layers are fed to the next Keras layer. This is often helpful in natural language processing.

For example, the regular RNN model with a GRU layer with $p = 10$ and $q = 32$ has $3 \times 32 \times (10+32+1) = 4128$ parameters for the GRU layer and $32 + 1 = 33$ parameters for the output layer.

```
model <- keras_model_sequential() %>%
  layer_gru(units = 32, input_shape = list(NULL, 10)) %>%
  layer_dense(units = 1, activation = "sigmoid")
summary(model)
```

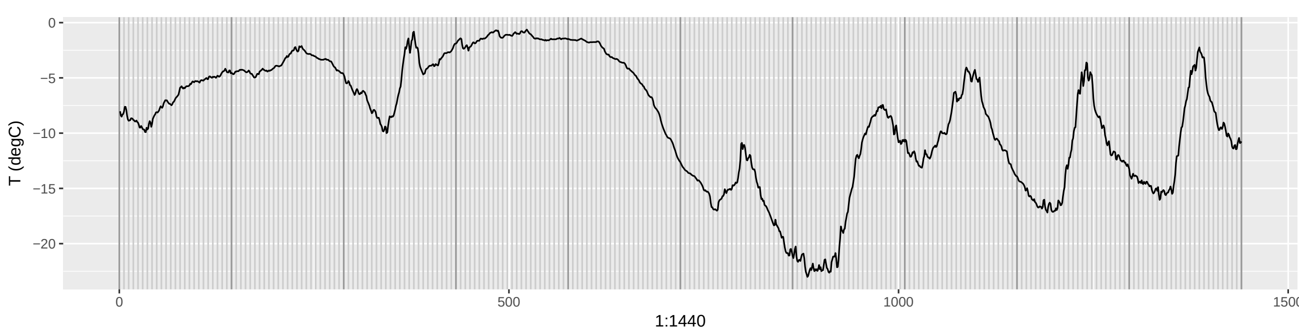
When the GRU layer is used bidirectionally, with the same $p = 10$ and $q = 32$, the model has $2 \times 4128 = 8256$ parameters for the bidirectional layer and $64 + 1 = 65$ parameters for the output layer.

```
model <- keras_model_sequential() %>%
  bidirectional(layer_gru(units = 32), input_shape = list(NULL, 10)) %>%
  layer_dense(units = 1)
summary(model)
```

14.5 Example: Temperature forecasting (many-to-one)

https://tensorflow.rstudio.com/blog/time-series-forecasting-with-recurrent-neural-networks.html

In this weather time series dataset, temperature and 13 other relevant variables (e.g., air pressure) were recorded every 10 minutes for 8 years from 2009 to 2016 in Jena, Germany. Below are the temperatures in the first 10 days of 2009. There are $6 \times 24 \times 10 = 1440$ values.



The authors want to predict the temperature 24 hours from now given the data in the last 10 days (temperature + 13 other variables). Instead of using the full 10 days data as the input, they sampled hourly (i.e., took every 6th observation; $k = 240$). In a FFNN, there would be a total of $240 \times 14 = 3360$ input neurons. For the RNN model below, the GRU layer has 240 unfolded layers, each with a 14-vector as the input. This model achieves a mean absolute error (MAE) of 2.35°C (4.23°F) after denormalization.

```
library(keras)
model <- keras_model_sequential() %>%
  layer_gru(units = 32, input_shape = list(NULL, 14)) %>%
  layer_dense(units = 1, activation = "relu")

summary(model)
```

14.6 Milestones of ANNs:

- Backpropagation (Rumelhart et al. 1986)
- RNN was developed (LSTM; (Hochreiter and Schmidhuber, 1997)
- MNIST dataset (1998): 70,000 28×28 images of handwritten digits, 10 classes
- CNN was developed and applied to MNIST (LeNet; LeCun et al. 1998)
- Natural language processing (NLP) (Bengio et al., 2003)
- Netflix Prize (2006–2009)
- CIFAR-10/CIFAR-100 datasets (2009): 60,000 32×32 images, in 10 and 100 classes.
- ImageNet dataset (Fei-Fei Li): Initially 3.2 million images in 5,247 "synonym sets" or "synsets". Now over 14 million images. 1000 synsets have SIFT (scale-invariant feature transform) features.
- ILSVRC (ImageNet Large Scale Visual Recognition Challenge) (2010–2017): **The catalyst for the AI boom**.
  - **AlexNet** (Krizhevsky, Sutskever, Hinton, 2012) **brought DL into the mainstream**. It won the 2012 ILSVRC (top-5 error rate 15.4%, same below; 11x11 filters). They used ReLU rather than the conventional tanh function, and introduced dropout layers to overcome overfitting.
  - **VGG Net** (Simonyan and Zisserman, 2014) won the "classification+localization" category of the 2014 ILSVRC (error rate 7.3%; 3x3 filters). Showed that simple deep structures work for hierarchical feature extraction.
  - **GoogLeNet/inception** (Szegedy et al., 2014) won 2014 ILSVRC (error rate 6.7%). Introduced the inception module, with CNN layers not stacked up sequentially. Later improvement: BN-inception-2 (error rate 4.8% with batch normalization) (Ioffe, Szegedy, 2015), inception-3 (error rate 3.6%) (Szegedy et al., 2015).
  - Microsoft **ResNet** (He et al. 2015) won 2015 ILSVRC (error rate 3.6%). Introduced residual block to reduce overfitting.
- Generative adversarial nets (GANs) (Goodfellow et al. 2014)
- AlphaGo (2016), AlphaGo Zero (2017), AlphaZero (2018)
- 2018 Turing Award to three pioneers in deep learning: Geoffrey Hinton, Yoshua Bengio, Yann LeCun.

Below is a figure from a 2017 article on the impact of the ImageNet dataset and the ILSVRC: The data that transformed AI research — and possibly the world. It shows the great performance improvements made every year between 2012 and 2015.