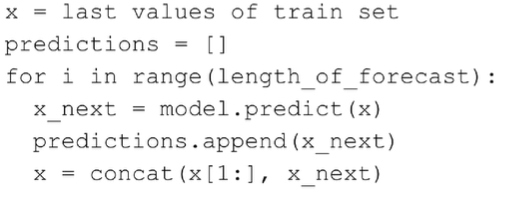
1. Sequence data

* Sequence data: text, speech, financial data / stock returns
* Time series:
* Any continuous-valued measurement taken periodically
* E.g.: stock price
* E.g.: Airline passengers
* E.g.: Weather
* Weather is a dynamical system (e.g. chaos theory, butterfly effect)
* E.g.: Speech / Audio
* Speech recognition
* E.g.: Text
* We don’t have to treat text as a sequence
* ML: treat as a bag-of-words (problem: lose info about the order of the word)
* What is a sequence?
* Usually, we think of 1D time series signal
* Linear regression – line of best fit
* Extend the concept to multiple dimensions
* For non-sequential data: NxD matrix
* Shape of a sequence:
* 3D array of size N x T x D
  + N = # samples
  + D = # features
  + T = # time steps in the sequence
* E.g.: Location data – model the path employees take to get to work, record GPS data from their cars
  + N: one sample – one person’s trip to work
  + D = 2: GPS record (latitude, longtitude)
  + T = number of measurements taken from start to finish of a single trip
    - E.g.: 30-min trip, measurement taken every second -> T = 30\*60 = 1800
* Variable length sequence?
* Each person take a different amount of time to get to work?
* In Tensorflow/Keras, we deal only with equal-length sequences
* E.g.: stock prices
  + D = 1
  + Suppose we use a window of size T = 10 to predict the next value
  + N = number of windows in the time series
  + Convolutional arithmetic:
    - If we have a sequence of 100 stock prices, how many windows of length 10 are there? 100 – 10 + 1 = 91
    - If we have a sequence of length L, and a window size T, then there are L-T+1 windows
  + what if we measure the stock price every day for 500 different stocks? S&P500
  + D = 500
  + We can still look at time windows T=10
* A single sample will be of size T x D = 10 x 500
* E.g.: Neural interfaces:
* D = # electrodes in the brain
* At each time step, electrodes measures voltage
* Predict the letter you want to type, using 1 sec. of measurements
* Sampling rate 1 sample/millisecond
  + Note: these samples are not the same as the N samples of (input, target) pairs
  + This is just signal processing terminology
* T = 1000
* N = # letters you/test subjects tried to type
* D = # electrodes
* Why N x T x D? see slides
* Visualize the data in your mind
* Variable length sequences:
* Pretty common scenario – e.g. sentences
* Some sentences have 1-2 words, some have 100
* In the past, we used variable length sequences -> complicated
* Inefficient data structures
* N x T x D – single array – Numpy arrays are fast
* Inefficiency of constant length sequences
* In Tensorflow & Keras, we use constant length sequence (unless using custom code)
* If variable length sequences (e.g., sentences) -> pad each sequence shorter than the longest sequence with 0s
  + E.g.: sentence with 1 or 2 words still takes up 100 blocks of memo
* Trade off

1. Forecasting

* Many do this in a way that looks nice >< results don’t make sense
* Predict the next values of a time series
* Number of future steps we want to predict – horizon
* E.g. predict demand for 3-5 days for products manufactured in a factory
* E.g. weather – hourly weather for 7 days (7\*24 = 168)
* We don’t usually want to predict just one step ahead
* Where are the RNNs?
* In ML (especially industry) – start with the simplest approach possible
* Simplest way to forecast a 1-D timeseries?
* Linear regression
* Most time series analysis involves only linear regression
* If data is N x T x D, but linear regression expects N x D, how can it work?
* For a 1-D time series -> D = 1 (superfluous dimension)
* N x T array contains the same data
* Just flatten the N x T x 1 to N x T
* Pretend T is D
* How does the data look?
* Time series of length 10
* Predict the next value using past 3 values
* Input matrix will be of shape N x 3, target (Y) will be of shape N
* N = 10-3+1 = 8???? -> where is 11th value?
* What does the model look like?
* Autoregressive (AR) model
* How do we forecast?
* Why not just plug in X\_test into model.predict(X\_test)?
* Predictions for x11, x12, x13, …
* No!
* Remember, we want to predict multiple steps ahead
* If we want to predict 3 days ahead – e.g. use (day1, day2, day3) to predict (day4, day5, day6)
* Before:
  + We used (day1, day2, day3) to predict day4 -> ok
  + We used (day2, day3, day4) to predict day5 -> not ok (we don’t know day 4 yet)
  + We used (day3, day4, day5) to predict day6 -> not ok
* In order to predict multiple steps into the future, we must use our own earlier predictions as inputs
* Because we must use our own predictions, we can’t just do model.predict() in one step



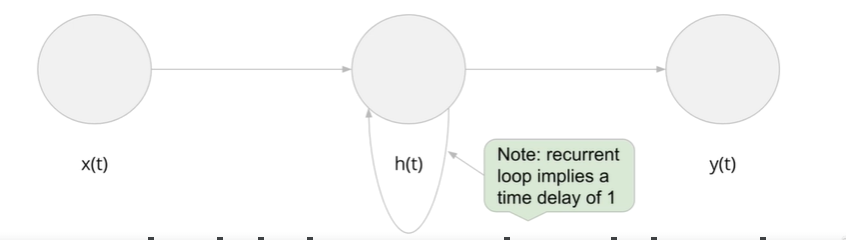
* All ML interfaces are the same

1. Proof that the linear model works (sine wave series)

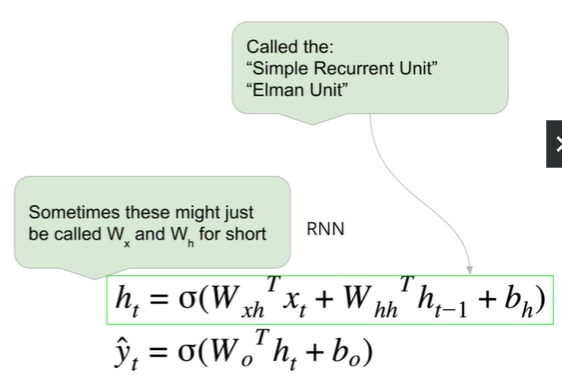
* Perfect forecast for a sine wave?
* How is this possible with a linear model?
* AR(2)
* It is possible to perfectly predict a since wave using only 2 previous values
* AR(2) model
* No bias term needed
* Sine wave as a function of time:
* We used the same equation to create the data
* Np.sin(0.1 \* np.arange(200))
* Np.arange(200) is all the different values of t
* 0.1 is the angular frequency
* Plug in the sine function
* Replace with
* Note: Fibonacci is another recurrence relation where the weights are 1
* Multiply out
* Trigonometric identities:
* Move term to the left hand side:

1. Recurrent neural networks

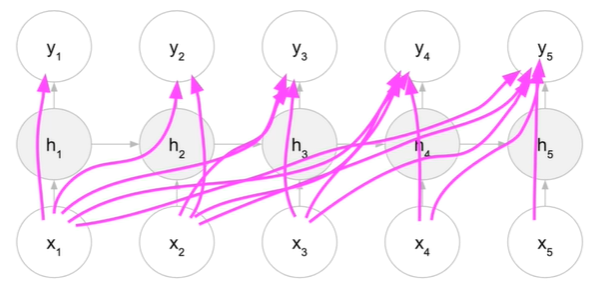
* Start with the dumb-as-possible approach
* What if D > 1? Multidimensional timeseries
* Why not just flatten this too?
* Flatten your T x D time series into a single feature vector
* E.g. 5 electrodes recording 100 time steps -> 500-length vector
* Full matrix multiplies take up space
* D = 100, T = 10000 -> T x D = 1 million
* Just like with CNNs, RNNs take advantage of the structure
* ANN is the most general – connects every input to every feature in the next hidden layer
* How can we exploit the structure of the sequence?
* Start with an ANN:
* How can we inspiration from forecasting?
* Make it an RNN:
* Make the hidden feature (hidden state) depend on previous hidden state (in addition to input)
* Linear regression forecasting model: output is linear function of inputs
* Now: hidden state is a nonlinear function of input and past hidden state
  + Nonlinear function = neuron



* RNN equation:
* Note: typically use 1 hidden layer (not 100 like a CNN)
* is a nonlinear function of and
* Helpful subscripts
  + x = input
  + h = hidden
  + o = output
  + xh = input to hidden
  + hh = hidden to hidden



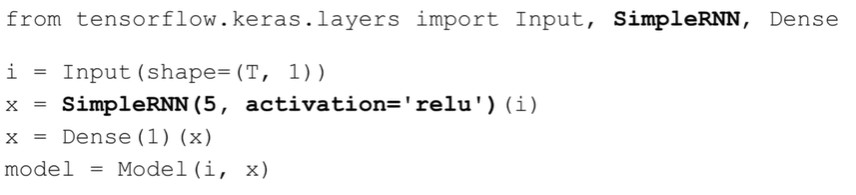
* How do we calculate?
* How do we get the output prediction?
* Given
* First step:
* What is ?
* Initial hidden state
* Can be a learned parameter (via gradient descent)
* In Tensorflow, it’s not learnable -> just assume it’s 0
* Now:
* Calculate the 2nd output
* We have
* Why multiple output predictions?
* Why do we have a prediction for each time step?
* If we are forecasting, we only want the next value
* For the problems previously described (we are looking for a single answer) -> discard the previous outputs
* Keep only
* Keeping multiple outputs
* Neural machine translation (both input and output are sentences)
* Need to capture prediction at each time point (step)
* Classification probability:
* For ANN or CNN, output is prob of each category given the input
* Machine translation is classification, because the target is a word (in the target language)
* What is the ‘given’?
* Unrolled RNN



* Relationship to Markov models
* Markov assumption: the current value depends only on the immediate previous value
* Not very powerful when trying to predict next word based on previous word
* An RNN language model
* An RNN will forecast the next word based on all previous words -> much more powerful
* Pseudocode
* Biological inspiration
* Recurrent neural circuits
* Hebbian learning: neurons that fire together wire together, and neurons that fire out of sync fail to link
* Calculating our savings (see slides)

1. RNN Code preparation

* Same forecasting exercise we just did, but with a simple RNN instead
* Steps:
* Step 1: load in the data
  + Same as before, but not the right shape for our RNN (N x T x D)
* Step 2: build the model
* Step 3: train the model
* Step 4: evaluate the model
* Step 5: make predictions
  + A little tricky – need to pay attention to shapes
* Load in the data
* Sine wave (with and without noise) – supervised learning dataset
  + Input: sequence of length T
  + Output: next value (scalar)
* Count from t = 0… len(series)-T
  + Since final target should be at len(series)-1
* Linear regression expects 2D input
  + N x T
* RNN expects 3D input
  + N x T x 1
* Instantiate the model:



* Training and evaluation: same as before
* Making predictions
* Input shape will be N x T x D, output will be N x K
  + N=1, D=1, K=1
* A single time-series input will be a 1D array of length T



1. Paying attention to shapes
2. GRU and LSTM

* Modern RNN units
* LSTM – long short-term memory
* GRU – gated recurrent unit -> a simplified version of the LSTM (less params and thus more efficient)
* Why do we need these at all?
* Why do we need fancy RNNs?
* Consider again the vanishing gradient problem
* The output prediction is a huge composite function, depending on
* appears several times (once at each step)
* We’ll need the gradient of for all t = 1, …, T
* The final gradient will be a function of all these individual gradients
* Consider how deeply nested each term is
* is the most deeply nested
* What do we remember about ANNs?
* ANN is a big composite function -> turn into multiplications in the derivative (chain rule)
* The more deeply nested, the more multiplications
* RNN are vulnerable to the vanishing gradient problem
* The further back an input is, the more its gradient vanishes
* The simple RNN can’t learn from inputs too far back
* Just use ReLU?
* Unfortunately, not so simple
* In Deep learning, researchers have discovered that GRUs and LSTMs are more effective

1. Gated Recurrent Unit (GRU)

* Same ‘API’ as the SimpleRNN

A diagram of a simple and simple process

Description automatically generated

* Diagrams vs. Equations

A diagram of a flowchart

Description automatically generated

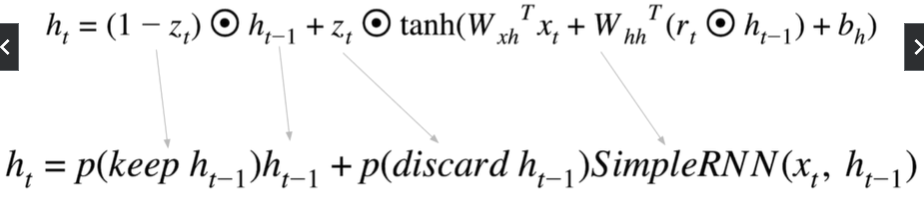
-> **Update gate vector**

-> **Reset gate vector**

-> **Hidden state**

* – vectors of size M
* M is a hyperparams (number of hidden units / features)
* This implies the shape of all the weights
* Any weight going from x(t) is D x M
* Any weight going from h(t) is M x M
* All bias terms are of size M
* **What is the GRU doing?**
* **z(t) = update gate vector**
* should I take the new value for h(t) or should I keep the old value h(t-1)?
  + Vanishing gradient means the RNN forgets things in the past
  + Now, we can explicitly remember the previous h(t-1) (if z(t) small)
  + If z(t) -> 1, then forget h(t-1)
  + Return of the neuron

* Logistic regression
* Sigmoid is not usually treated as a hyperparam
* We want z(t) to be in range (0, 1) -> binary classifier, telling us which thing to choose to get h(t)



* **r(t) = reset gate**
  + Just another neuron – ‘switch’ to remember / forget h(t-1)
  + r(t) is between (0, 1)
* **GRU summary:**
* Same API as simple recurrent unit
* Output is h(t), depends on h(t-1) and x(t)
* Has gates to remember / forget each component of h(t-1)
* SimpleRNNs have no choice but to eventually forget, due to the vanishing gradient
* We use binary classifiers (logistic regression neurons) as our gates

1. LSTM

* SimpleRNN has problem learning long-term dependencies
* The hidden state becomes the weighted sum of the previous hidden state and new value (allowing you to remember old state)
* Controlled by gates which are like binary classifiers neurons
* When GRUs first came out, there was no clear winner >< New research in favor of LSTMs
* Experimental results should guide best practices
* **LSTM – like the GRU but with more state vectors and gates**
* Not exactly the same API
* **LSTM returns 2 states**:
  + **Hidden state** h(t)
  + **Cell state** c(t) (usually ignored)
* LSTM unit in Tensorflow outputs h(T), but can also optionally output c(T)
* Also means you **need 2 initial states**:



* **LSTM equations**

-> **Forget gate**

-> **Input / update gate**

-> **Output gate**

-> **cell state**

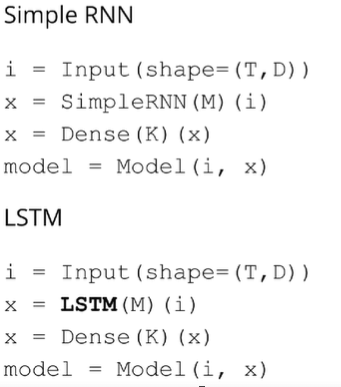
-> **hidden state**

Where:

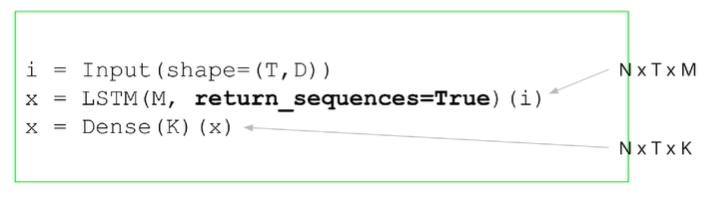
* **Simplified**:
* f(t) = neuron (binary classifier)
* i(t) = neuron (binary classifier)
* o(t) = neuron (binary classifier)
* c(t) = f(t) \* c(t-1) + i(t) \* SimpleRNN
* h(t) = o(t) \* tanh(c(t))
* Note: tanh() is usually kept even though it’s possible to change

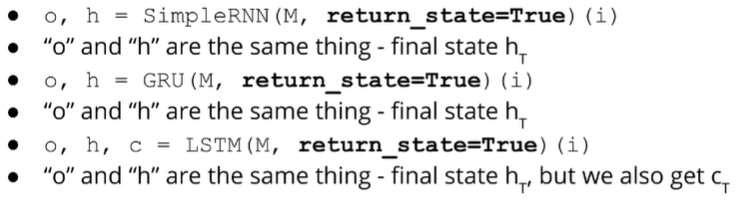
If you change it, the LSTM unit won’t be GPU-compatible

* In code:



* **Options for RNN units**
* For each , we will calculate
* We’ve seen that the SimpleRNN, GRU, LSTM will return by default
* We may want all of





1. RNN for image classification

* Dumb-as-possible approach:
* Tabular data
* These features made up the input feature vector
* For images, we just flattened the pixels and made it a vector
* We thus pretend each pixel value is the answer to a survey question
* Time series -> pretend each value in the sequence is the answer to a survey question
* How else can we use our imagination?
* A multidimensional time series is a T x D matrix (2D)
  + Each column is a time series (top -> bottom)
* Consider a black and white image
  + Also a 2D H x W matrix
  + X[i,j] is the pixel intensity at row i, column j
* Since they are both 2D, can we simply pretend an image is a time series?
* Like an image scanner – look at each row of the image at a time
* Code preparation
* Step 1: load in the data
  + X is of shape N x T x D
* Step 2: Instantiate the model
  + LSTM -> Dense(10, activation=’softmax’)
* Step 3: Fit the model etc.
* Side note: try global max pooling too

1. Stock return predictions using LSTM

* Lesson 1: one-step prediction on stock prices is misleading and also unconventional
* More conventionally predict stock return
* Return intuition
* What does it mean for something to be 20% off?
* Intuitively, if something costs $100 and is 20% off, you’ll pay $80
* 3rd model
* Make use of all data: open, high, low, close, volume (D=5)
* Predict whether the price will go up or down (equivalent to predicting whether the return is +ve or -ve)
* ML intuition: regression is harder than classification

1. Other ways to forecast

* One-step forecasts can seem artificially good
* Iteratively building a multi-step forecast can lead to poor results, even on simple problems (like the sine wave, even our perfect model breaks down eventually)
* Practicality:
* The constraints of your project/company/data are what matters
* Not necessarily wrong to predict only 1 step ahead -> has to make sense
* E.g. forecast weekly or monthly sales for next year
  + Multi-step forecast
* E.g. predict load on your website tomorrow so you can spin up some Amazon machine instances
* Benchmarking
* Use a baseline model (e.g., naïve forecast)
* Naïve forecast = dumbest thing possible
  + Just predict the last value
  + Zero param model
* If you ever study finance:
  + Stock prices closely follow a random walk
  + The naïve forecast is the best forecast
* Multi-step predictor
* Classical statistical methods like ARIMA build a one-step predictor, then iteratively apply it to forecast multiple steps ahead
* Can a model naturally forecast multiple steps? Yes
  + E.g. 12 steps ahead -> Dense(12)
* Summary:
* Big mistake: pretend you are predicting the future when you’re really predicting only one step ahead
* You can’t use data from the future to predict multiple steps in the future
* In a random walk model (very close to stock prices), a naïve forecast is the best forecast
* Easy to modify a model to naturally make a multi-step forecast, just have multiple outputs