SYSTEMATIC TRADING NOTES

Alessandro Muci
alex.muci@gmail.com

Abstract

Notes on systematic trading, starting from general considerations to single strategies ...

This is a very rough and incomplete DRAFT.
TODO: Fix code tabs

Contents

Chapter 1

Introduction [TODO: SPLIT the chapter]

blah blah blah

1.1 The two approaches: rationale

Why we need rules-based (aka mechanical) trading systems
. . . [TO BE COMPLETED] . . .

1.1.1 Human biases

Human beings are no devoid of emotions that get into the way of rational behaviour. Homo economicus, who is supposed to be the rational and driven by greed (as economists or Hollywood films would like us to believe), is actually an envious animal caring more to keep us with the Jones that to better himself. In the first case, we expect men to maximise their wealth given some risk constraints (think of utility maximisation or modern portfolio theory, i.e. men improving their conditions independently of the others - which is the rational thing to do), yet men seem to care more to be better off than their peers or, in other words, the strive to achieve status (better being prince in hell than one of the angels in heaven) irrespective of their

actual well-being (in other words, they prefer being the lucky one to have a job in a favella than just a millionaire in a neighbour of billionaires).

Moreover, men are weighed down by a pletora of behavioural biases of which overconfidence is one of the worst... INSERT SOME EXAMPLE: simple systems are better in diagnosis than most doctors, ... not immune to the passions of the day, seeking an objective truth that will triumph over obstacles.

1.1.2 Epistemic limits

In addition, even if men were perfectly rational there still remains the issue of what knowledge is really available available to us. In other words, we are subject to a narrow epistemic horizon: we are not able to forecast, with a reasonable degree of certainty, what will be the results of future economic or social policies. We leave in a probabilistic world, unlike the Newtons deterministic model of reality that gave 18^{th} century scientists the dream of a perfectly predictable world. Were we leaving in a clockwork universe then - as Laplace asserted in 1814 - we would be able, in principle, to predict everything for all time. With ever more powerful computers, that 'in principle' might have seemed closer to 'in practice', but complex systems (like weather, markets or animal populations) turn out to be extremely sensitive to tiny variations in initial conditions - which are beyond our ability to measure.

In short, the unpredictability and fundamental uncertainty of many aspects of reality makes perfect predictions à la Laplace impossible not just in practice but in theory too.

1.1.3 Deduction vs. induction

There are two approaches not just in finding trading ideas/signals but to advance our knowledge in general:

- The more traditional one is the Galilean (scientific) approach: one formulates an idea or hypothesis and then test it empirically, going back to a new idea if first hypothesis rejected.

- The other path is to start the other way around (thanks mostly to recent progresses in machine learning), i.e. directly from the data by letting a computer figure out patterns that humans cannot possibly see (data-mining).

In this chapter, we follow the traditional approach because easier and more robust. The astonishing gains in machine learning, which have undoubtedly helped in some areas, are still too shallow for complex financial markets. Machine learning methods can be used for some areas ... [TO BE COMPLETED] ...

The system we propose in 1.6 is simple, objective and transparent (computer can run it), based on underlying economic ideas (vs. data-mining). We explicitly try to avoid the three excesses of bad systems: over-fitting, over-trading and over-betting.

1.2 Achievable Sharpe

First, let's see what it is achievable and what is simply wishful thinking (i.e. over-fitting, results of typos [report errors found by Zakamilin], ...).

Let's start with **equity**. Equity markets have usually had a premium (from earlier economists' studies) of around 6% - based on the USA equity indices, arguably the best performing in the world - that has been more recently revised to a less optimist 3-4%. That implies a Sharpe ratios of around 0.25 (i.e. 3.5%/15%). For single shares, given higher volatilities, one can expect something lower, say around 0.15.

Diversified baskets across asset classes have historically fare better with Sharpe ratios of 0.3-0.4.

Moving from buy-and-hold to dynamic systems, one can go from **single asset** rules-based strategy with 0.5 Sharpe ratios to **multi-asset** systematic investment of 0.8-1.1.

Sharpe ratios above 1 are rare, even for sophisticated investors (and the realm of capacity-constrained[1] High Frequency traders). Indeed very high

[1] That means that these strategies are generally not scalable and therefore not offered to the wider public but kept for own use.

SR are the result of negative skew strategies (akin to selling insurance) that will blow up when the cycle turns

1.3 Trading rules

When designing trading systems, we would i. avoid to mess around a system with discretionary intervention (accepting our behavioural biases), which ironically requires either iron discipline or extreme laziness, ii. try to incorporate redundancy (in rules, ...) because of our limited and uncertain knowledge and iii. follow a deductive (idea-first, data later) approach.

To make our implementation easier (in terms of computer coding, actual testing, future strategy additions, ...), we follow a modular approach by keeping separated the various parts of the system.

Rolling windows or 5 years rather than using part (e.g. half or 2/3rd) of the data for calibration/optimisation and the rest for testing.

Avoid too many variations because then it becomes difficult to select [get Do Prado paper].

How long should history be: T-test for SR, the lower the SR the longer history needs to be:

Table 1.1: Sharpe ratios vs. years

Sharpe ratio	0.2	0.4	0.5	0.7	1.0	1.5
years	45	33	20	10	6	3

how long to decide that a rule is better than another competing one? The less correlated and lower advantage (in terms of SR) the more difficult (typically > 30-40y)

Keep it simple: use rolling windows and do not discard rules/ be aware of skew / measure performance vs. benchmark (to avoid confusing strategy returns with secular trends. e.g. last 30y bull bond market).

Trading rules and stop losses should *solely* depend on mkt volatility. Statements like 3% of stop loss may be either too tight or too loose depending on the instruments

Portfolio

1. corner solutions, unstable: typically use of portfolio optimisation 2. but easier for trading strategies since mostly are 'vol standardised' and 'long only' (no short-sales allowed) 3. equal weights? same return and same vols (not just same SR), same correl

"shrinkage" $->$ reduce uncertainties (see Garlappi et al ...)

Better solutions:

- bootstrapping (brute force): taking avg of multiple optimisations to filter out the noise;
- Leverage Model [`MS thesis in 'Books' folder`];

1.3.1 Profit sources

Source of profits are (`ref to Illmanen's book`):

- traditional risk-premia (duration, equity risk, peso problem, ...)
- leverage (good leverage, 'value' vs. 'growth')
- barriers to entry (private equity, High Frequency Trading)
- non-opportunistic (central banks intervention, window dressing)

Trading styles:

- blah blah blah ...
- blah blah blah ...

1.3.2 Instruments

Real-world constraints:

- **market data availability** (cheap and rentable databases: Quandl and QuantGo - the latter for trading systems living in the cloud): on the risk of using EOD (end of date) to backtesting system (EPChan blog: http://epchan.blogspot.co.uk/2015/04/beware-of-low-frequency-data.html, that is the difference between auction closing prices and

consolidated prices). Even when a strategy trades at low frequency, maybe as low as once a month, we often still require high frequency ticks-and-quotes data to backtest it properly

- **minimum size**: e.g. JPY Government futures too big vs. most retail accounts to even trade one lot (USD 1.5 mln).
- **avoid low vol** instruments, particularly if pegged (e.g. CHF/EUR gyrations in Jan 2015 and following unfulfilled/dishonoured quotes, brokers' defaults, etc.)

Style choices:

- hold max number of instruments given account size (trageted volatility and underlying min lot size);
- correlation (obvious): the less correlated, the better;
- cost & liquidity: cheap instruments (e.g. futures) if higher frequency trading (< few days);
- negative skew instruments must be managed with care: they make sense coupled with momentum, but exposure should be limited. Always avoid pegged instruments or whenever political rather than economic decisions are the ultimate drivers (e.g. all pegged currencies).

Access:

- Exchange vs. OTC: the first better - e.g. again CHF/USD re-quotes or absence of available quotes in Jan 2015 that brought more losses to retail accounts. Even for FX may make sense to preference ccy futures over OTC spot transactions.
- Cash or derivatives: better the second, since it offers straightforward leverage but need to consider costs & lot min size (vs. account size): for small retail accounts ETFs may be the sole possibility vs. futures (better for bigger account sizes, > USD 100k).

1.3.3 Signals

Signals must have the following features:

- quantities...: not just binary (i.e. just all-in long or short), but they must produce *how much* to buy or sell. Besides cheaper to trade (not need to flip positions), it allows to take into account the strength of a signal. Most published systems seem to be of the all or nothing types, hence ignoring the limits of our knowledge (see sub-section in 1.1) ...;
- ... on a consistent and capped scale...: have a scale that represents strengths of the signal in a consistent way, from flat to weak, normal, strong and very strong and both long/short (e.g. -2, -1.5, -1, -0.5, 0 flat, +0.5, +1, +1.5, +2), but capped at +/- 2 (no signal must be so strong to bet all on it[2]);
- ... proportionally to expected scaled return (e.g. Sharpe Ratio): to account for the underlying instrument risk (e.g. volatility), making the rule usable across instruments (no asset specific) and periods too;

In section 1.6, we would present two popular sets of signals as an example.

Combining signals

Frequency and costs

TO BE EXPANDED:

- Timeframe of trading: from microseconds to days to weeks
- costs: how to standardise the costs (number to subtract from a strategy gross Sharpe ratio to produce the net one)
- capital impact

1.4 Money management

This is usually the part most overlooked by novice traders, who tend to focus too much on the trading signal variations overlooking to determine the risk they like vs. the actual running. We split the money management between

[2]Cap is to limit negative surprise, particularly following low vol periods. Morevoer, if signals were Gaussian distributed, we would expect them stronger than 2x less than 5% of times - no signal hence warrants

its two main components: the amount of the risk we would like to run 1.4.1 and the actual position size of each instruments.

1.4.1 Risk management

how much we are prepared to lose and
our system expect to experience

Stop losses: no need for those whith rule re-calibration
quote paper with stop losses
entry-system needs to be re-calculated, that would serve as an exit point.

Defining Risk

Risk is different things to different people and can be defined on different levels. It is first of all a question of time horizon: one day, one year or a lifetime? After all, all systems are bound to fail in the long-run[3]. And, secondly, of quantum: drawdown (peak-to-valley), risk of ruin, etc.
Predicatable and non $->$ Donald Rumseld: known knows, known unknows, unkown unkowns.
Start from use of annualised standard deviation or volatility: standard deviation is measured in percentage or money terms, and over different time period. We call **volatility** here, as mostly in the finance literature, the annualised (via square-root rule) standard deviation of daily log-returns.
Annualised vs annual: implicitly assumption of zero correlations.
annualised volatility is not the average or maximum expected to lose over 1 year
Everybody knows that volatility is not risk, but our assumption here is just that vol is proportinal to risk
consider Skew (i.e. third moment) and kurtosis (fourth moment)
To recap: annualised (percentage) volatility of returns.

[3] 'Even the Roman Empire did collapse...'

Risk targeting

Why volatility

Most books talk about percentage of capital, e.g. 1% of available capital: we don't because otherwise we are ignoring time-dimension/timeframe, e.g. 1% of risk over 1 day or over 1 week holding period? Thinking in terms of volatility (annualised standard deviation) makes our life easier. It can be translated in money terms by simply multiplying it by the trading capital
Other moments of risks will be dealt differently. INSERT TABLE
We are here ignoring where the investment is made, or in other words that 100k invested in the Eurostoxx50 is not as risky as 100k invested in European goverment bonds - more on this in the next section.

Setting a volatility target

[overall risk of all trading sub-stystems, it must be based on available capital (account size) and risk preference (vol).]
Risk appetite: 100k capital and 50% vol target means we are going to risk 50k. That gives an idea given of our expected returns (given excess return = vol * Sharpe) and expected losses vs. time horizon (e.g. assuming a Gaussian distribution).
Leverage: given the underlying instrument volatility and our own chosen risk appetite, it becomes clearer our targeted return (or conversely if achievable). [Enter here the Skew and other moments issue with too much of a leverage].
Volatility drag: geometric average vs. arithmetic [approx geom avg = arithmetic avg - 0.5 x $StDev^2$] - this is sometimes one of the explanation given for the low volatility shares better performance over long periods.
The targeted risky capital is simply:

$$C_t \times TargVol \tag{1.1}$$

where $TargVol$ is the volatility (e.g. 20%). Now, at the risk of stating the obvious, the targeted risky capital is not the maximum amount we may lose[4] but just a working assumption for risk.

1.4.2 Setting instruments weights

Lot value

In order to differentiate obvious instruments (e.g. one equity share value is the price of one share) with less immediate ones (e.g. one gold futures whose price is expressed for Troy ounches ... or futures quoted per barrel or bushel, with each contrat for a number of these ounches/barrel/...). One simple and natural choice is to express the value of each instrument in terms of the amount you are exposed to in your chosen currency. In other words, the face value translates in referenced currency the size of each futures contract (whether in dollars or Euro, in ounces or barrels and multiple of thereof).

$$LotValue_t = P_t^{traded} \times FX_t \times mult \tag{1.2}$$

where

- P_t^{traded} is the quoted price of the traded futures;
- FX_t is the exchange rate epressed as number of units of the currency in which the account/capital is denominated (e.g. EUR) for 1 unit of the currency of the traded contract (e.g. USD - hence, in our example, $FX_t = \text{USDEUR}_t = 1/\text{EURUSD}_t$);
- $mult$ is the number of units embedded in the futures contract (e.g. (a) 1,000 in regards of WTI futures, being the quoted price for 1 barrel and the contrat units for 1,000 barrels and (b) 10 in respect of Eurostoxx50 index futures, being each index point worth 10 Euros).

[4] Even assuming a benign normal distribution of returns, we expect our annual returns to be outside the region -20% and +20%/ around the average return

Standardise instruments value

We now want to take into account the embedded riskiness of each instument...
When measuring past volatility is important to decide which windows? trade-off between noise (and more frequent rebalancing) and relevance. In practice, the choice it typically between 20 days and 6 months.

Assuming an investment in a single instrument, we buy/sell the following number of gross units:

$$gross\_units_t = \frac{C_t \times TargVol}{LotValue_t \times \sigma_t} \tag{1.3}$$

Note the time subscripts in the equation imply that our units will change even assuming constant signals.

1.4.3 Putting all together: signals & instruments

Adjstment for diversifications in two consequent steps:

- Signal variations: the easiest option is a linear combination corrected for the diversification benefits ($= 1/\sqrt{WCW^T}$ where C is the weekly correlation matrix across signals), another option used by actual hedge funds is a response function [see Richard Martin or MAN-AHL], and then
- Instruments: ...

Assuming a portfolio of instruments, the (number of) units for each is:

$$\begin{aligned} units &= PortW_t \times Sgnl_t \times gross\_units_t \\ &= PortW_t \times Sgnl_t \times \frac{C_t \times TargVol}{LotValue_t \times \sigma_t} \end{aligned} \tag{1.4}$$

where

$$Sgnl_t = \min(2, \max(-2, \frac{\sum_{i=1}^{...} sgnl_t^i}{\sqrt{WCW^T}})) \tag{1.5}$$

units is an integer (number of lots) hence it needs to be rounded.

1.4.4 Rebalancing

To control risk, as already noted above, one has to adjust the positions as needed (not simply periodically).
Rebalancing is due to a number of factors, besides changes signals and volatility one has to consider rolling futures contracts....

1.4.5 Futures optimal roll

Best periods to roll ...
never contemporaneous with signals ...

Rolling profits over

Regarding profits, obviously using the same number of lots (contract) not matter our (every changing) account size is clearly sub-optimal.

Two common are: begin

1. Fixed Fractional money management: lots traded (units) increase linearly with account size;
2. Fixed Ratio money management: lots increase with the square root of the account size (grow less and less as account size grows, assuming positive Sharpe)[5]

If the goal is to maximise our final wealth (i.e. the geometric return), then one should roll-over all the profits and follow the first scheme (or similar) above. This makes sense, but it completely ignores the uncertainty around our expected Sharpe ratio. As the returns grow exponentially, so do the drawdowns.
INSERT DEMONSTRATION
A more sensible alternative, we believe, is to reinvest only part of the profits. The above formula - drawdown growth proportionally to square root of

[5]The first was proposed by Ralph Vince and the second by Ryan Jones, both around the same time in early 1990's.

profits - suggests that one way to control drawdown is a scheme like the second one above.

In addition, there are a few more practical reasons to roll profits only *partially* :

- relying on some returns of the trading to live off;
- paying taxes (if trading does not take place in an efficient wrapper);
- lock-in profits (see academic papers on the desirability of such features ...).

The first two are self-explanatory, and the third one is just another side of the coin of the previous arugment. It is important also because:

- easier to stick to the system and avoiding meddling with it by reducing regret and
- robustnessness

In short, our preferred rebalancing rules for profits is to roll-over the **square root** of the profits. If you pay taxes and leave off trading returns, then it is probably best not to roll over most of the profit[6].

1.5 Backtesting

Backtesting is important to evaluate the performance of both strategies developed from scratch and those published. Profitability often rests on the implementation details that tend to be glossed over in articles/papers. Moreover you can improve the strategy and test it out-of-sample.

Importance of event-driven platform (vs. streaming ones, i.e. for-loop), particularly complex-event-processing: more complex to write for backtesting (vs simpler for-loop), but same code is then usable for live trading.

1.5.1 Common issues

- **look-ahead bias**. It is essentially a programming error that can invalidate a backtesting (but not, obviously, a live trading), e.g. using a

[6] Obviously, all losses must reduce our number of lots

day's high price for signal for same day.

- **data-snooping**, another name for over-fitting (too many parameters and/or non-linear complex models) to in-sample data. Solutions are: out-of-sample, cross-validation, using simple (e.g. linear) models with few parameters (Occam's razor), equal-weighting signals turned into z-scores, or rank z-score (e.g. Joel Greenblatt's magic formula).
- **share splits and dividend adjustments**. Need to adjusted for such events (drops due to divs or jumps for stock splits).
- **share survivorship bias**. Make backtesting wrong. E.g. if you invest in stocks that drop the most in the previous day and hold it forever, returns will be extremely negative in real life; but if your data miss delisted stocks backtesting is going to look great.
- **share short-sale constraints**. You cannot always locate and borrow a share to short. Sometimes (e.g. financial shares) that is simply forbidden by regulations.
- **share primary vs. consolidated prices**. E.g. Open or close are the first or last among various venues, but for proper backtesting you need primary exchange (since that is what is going to happen in real life). Same for high or low, coming most likely from consolidated ones.Note new alternative uptick rule (since 2010, old version until 2007).
- **FX venue dependency** same as per stocks, plus bid-ask very much dependent on venue.
- **futures continuous contracts**. Futures expiry and need to be rolled over. Backtesting usually make use of continuous contracts: (i) same chain (e.g. all front) contracts are concatenated and (ii) price gaps at rolls eliminated (usually back-wards) in two alternative ways: (a) price adjustment (adding price difference between new and old contracts, $q_{t+1} - p_{t+1}$, to prices p_t on or before t) or (b) return adjustment (multiplying every price p_t by the ratio q_{t+1}/p_{t+1} on or before t). In the former case only the PnL ($q_{t+1} - q_t$) is correct, whilst in the second case only the return ($q_{t+1}/q_t - 1$) is fine. Depending on the strategy one needs to choose either the first (e.g. signals based on spreads) or the second (signals based on ratio/returns).

- **futures close vs. settlement prices**. Daily closing prices of futures provided by vendors are usually settlement prices, the latter calculated by exchanges even if there was no actual traded price close by. Most of the times settlement prices are those to be used for daily prices.

1.5.2 Hypothesis testing

Generally, this is made up of 4 steps:

- We compute a test static, usually a (finite) average of daily returns of the strategy.
- We then suppose that this average is zero on an infinite data set, our *(null hypothesis*).
- then we determine or suppose the distribution of the daily returns, with zero mean (based on null hypothesis),
- based on this null hypothesis distribution, we compute the *p-value*, i.e. the probability that the average daily returns are at least as large (or extreme) as the observed value - if the p-value is very small (e.g. 0.01) we can 'reject the null hypothesis' and conclude that the backtest avg return is 'statistically significant'.

The most important step is determining the distribution, e.g. Gaussian with zero mean and std dev equal to the sample one or determined via Monte Carlo (simulating either the price data or actual strategy PnL itself).

In the Gaussian case the test statistic is simply the sample average divided by the sample standard deviation (i.e. Sharpe/Info ratio) multiplied by square root of data points: $\mu/\sigma\sqrt{n}$. E.g. if daily Sharpe times sqrt of backtest days is above 2.326 (or 1.645) then the p-value is smaller or equal to 0.01 (or 0.05) for Gaussian distribution.

For the MC case, (i) we generate random daily returns (not daily returns of the strategy) - for example using scipy.stats.pearson3 we can simulate returns with mean, variance and skew of the actual daily returns (but not same correlations or dependence structure) - (ii) apply strategy rules to such random returns a number of times, say 10.000. If high proportion of simulated strategies show CAGRs (or means) equal to or higher than

backtesting (on actual data), it means that our strategy returns are merely due to luck (lucky past returns used in backtesting), i.e. the p-value will be too high and we cannot refuse the null hypothesis. Ideally we want less tha 5% (or even 1%) of simulated strategies to show higher returns: our trading strategy profitability should capture some subtle patterns or dependence of past prices, not just the first few moments of price distributions ().

A variation of the MC is to simulate the trades (rather than the data) with the constraints that number of trades and average holding periods are the same as per backtesting. Usually this is a weaker test.

Different ways to calculate the distribution will give different answers. That is to be expected. After all we really want to know the conditional probability that the null hypothesis is true given the observed statistic $t-rtrn$: $Pr(H_0|t-rtrn)$, but we actually just computed $Pr(t-rtrn|H_0)$, and rarely the two are the same!

1.6 Example: CTA strategy

We present a complete trading system based on two signals that, as mentioned in section 1.3.3, are expressed as capped and scaled quantities. In addition, they are complementary (lowly correlated): the first profits from either positive or negative trends, whilst the second from range-bounded markets.

The system, although simple, is able to capture around 90% of the returns of established CTAs if applied to 20-25 futures. Here we assume a buddy retail investos with far less money [...].

1.6.1 Futures

Unless one starts with a few millions, selecting which futures to trade is a trade-off between:

- striving for maximum **diversification**: the more, the smoother the equity line - we should aim to, at least, one instrument across 4-5 asset classes;

- **minimum size**: we would like to trade up to 4 lots of each instrument to express 4 levels of signal strenght (hence avoiding binary all-or-nothing decisions).

Assuming an account size of EUR 100k and a rather aggressive target vol of 40% [TO BE CHECKED - also check T-note delivery option][7], a possible list of futures is in Table 1.2.

Table 1.2: Futures selection

Contract name	expiry	roll time	notes
Eurodollar	16th (4+12)	every 3-months	none
10y T-note	front	25dd before expiry	delivery options
EuroStoxx50E	front	10dd before expiry	none
VStoxx	2nd	25dd before expiry	carry-optimal
MXP/USD	front	10dd before	none
Corn	Dec	4-month before expiry	none

Get the futures data and EURUSD rate from Quandl.com

Stitching futures

Futures expiry hence we have to close and enter a new one. A process that is called 'roll', since we roll our position from an expirying contract to a longer-term one. Since it costs nothing to enter into a futures contract, any PnL is given by the difference between entering and exiting prices - entering into a new futures does not cost the difference between previously exiting price and the new. Hence we want to eliminate the gaps between the price of the old and newly-rolled futures, a process called 'stitching' by traders, to obtain a more accurate depiction of the PnL of our trading (otherwise we will be implicitly assuming as perfectly forecastable all gaps).

First, we need to decide when we roll between futures, for example 1 month prior to expiration. More precisely: we exit a futures at the closing price on the last trading day of the last-but-one month before the futures

[7] One author, Rob Carver, suggests USD 250k with 20% vol target whilst another, Chenlow, USD 1mln for achieving some meaningful diversification at lower vols.

expiries, and at the opening price on very next trading day enter the next futures i.e. the contract having the second-earliest expiry date on that day. One way to stitch is backward: we start with the latest futures in our dataset, go back to the day where we exited the previous contract and shift the entire price series of the previous up or down such that, on that day, the closing price of the two futures is the same. We continue this process until we reach the beginning of our data set[8].

The backward-adjustment has the effect to introduce a spurious negative drift for futures in contango (and vice versa for those in backwardation), for longer periods past adjusted prices may even turn negatives. But such effect does not really matter since our adjusted continuous series are meaningless in themselves (we only care about *differences*) and used only as an useful construct to ease signals/performance computation. As such to calculate long (or short) trade performances we also have to store actual prices and use the following formula:

$$(-1) \times \frac{\text{back-adj exit price} - \text{back-adj open price}}{\text{actual open price}}$$

1.6.2 Signals

[to be completed ...]

Momentun

This signal captures trends (i.e. absolute, rather than relative, momentum) using, say, (exponential moving average) double cross-overs (1-pole filters):

$$Mom(fast, slow, Cap)_t = \max[-Cap, \min\left(Cap, \frac{S(fast, slow)_t}{Avg_t|S(fast, slow)|}\right)] \tag{1.6}$$

where:

- $S(fast, slow)_t = \frac{EWMA_t^{fast} - EWMA_t^{slow}}{\sigma_t \times P_t}$,

[8] An alternative, called forward-adjustment, is to do the reverse, by starting the adjustment from the fist futures in our dataset.

- $Avg_t|S(fast, slow)|$ is the rolling average of *absolute* values of $S(fast, slow)_t$ over the previous year;
- $EWMA(n)$ is the exponential moving average with a $\lambda = 1-2/(1+n)$ where n is the number of days;
- P_t is the front futures price at date t and σ_t its annualised price returns volatility over past 30 business days, and
- $slow = 4 \times fast$
- $Cap = 2$

Setting the fastest n equal to 15 days, we can easily combine three double cross-overs:

$$Agg\_Mom(fast, slow, Cap)_t = \sum_{i=0}^{2} \frac{1}{3} \times \frac{Mom(15 \times 2^i, 15 \times 2^{2+i}, Cap)_t}{\sqrt{WCW^T}} \tag{1.7}$$

Thus our aggregate Momentum signal (for each instrument) is simply the arithmetic average of the 3 double cross-overs, the signal will have a cap of +/-2 that has not be included in the above formula for ease of exposition. [to be completed... for arguments in weighed more the front and last cross-over see Hamil's paper of MAN AHL at page 9, e.g. 40% - 20% - 40%]

Momentum is positively skewed (frequent small losses and rather large gains: profitable also with losses more frequent than gains). **Note that binary signals have zero Skew** !!! [to be completed...]

Carry

Carry is the return one gets if futures price curves remain unchanged, and indeed also named roll-yield or roll-down ...
It is negatively skewed (aka selling vol or lending): one gets paid in good times for bearing larger losses in bad times (see Illmanen)

$$Carry_t = \frac{P_t^T - P_t^{T+1}}{P_t^T \times \sigma_t^P} Carry_t = \max[-Cap, \min\left(Cap, \frac{RawCarry_t}{Avg_t|RawCarry_t|}\right)] \tag{1.8}$$

where:

- $RawCarry_t = \frac{P_t^T - P_t^{T+1}}{P_t^T \times \sigma_t^P}$
- $Avg_t|RawCarry_t|$ is the rolling average of *absolute* values of $RawCarry_t$ over the previous year;
- P_t^T is the price on date t of the futures contract expirying at T and
- P_t^{T+1} the price on same date of the next futures (expirying at $T+1$)

For agricultural and energy commodities (in our case just Corn), in order to avoid well-known seasonality effects, the P_t^T is the front contract and P_t^{T+1} the contract expirying exactly one year later.

Intuitively, we want to be long curves in backwardation (positive carry signal) and short those in contango (negative carry).

1.6.3 Portfolio

50% per each signal.
Weights for instruments (to be based on sub-system correlations):

Table 1.3: Instrument weights

Contract name	weight
Eurodollar	15.0%
10y T-note	15.0%
EuroStoxx50E	25.0%
VStoxx	10.0%
MXP/USD	17.5%
Corn	17.5%

Procedure

The trading system [daily] procedure to put everything together . . . :

- Calculate gross units[9]: get (i) account value from the broker and, for each instrument, (ii) prices and (iii) FX (here just EURUSD) from quandl.com;
- calculate aggregate signal, $Sgnl_t$, i.e. 3 (capped) double-crosses and one (capped) carry indicator for each instrument (and further apply on top +/- 2 cap);
- update the units: multiply instrument weight in table 1.3 by aggregate signal (and by diversification multiplier (e.g. 1.75)), round the result to the nearest integer;
- compare updated units vs. current traded units, trade if (i) the absolute differences are above [] and/or (ii) we need to roll a futures.

1.6.4 Portfolio

Code implemented in Quantopian:

```
"""
A simple CTA-style strategy.

* Use two main signals of standard CTA:
A. TREND (with positive skew: large gains and more frequent smaller losses):
i. calculate 3 sets of moving-averages-crosses (15/60, 30/120, 60/240)
   for each continuous futures;
2. get the mean of these 3 MA-crosses and scale it by price vol
   (i.e. / (annualised return vol * price));
3. re-based such result by the mean of the absolute MA-cross
   values over the past year.

B. CARRY (with negative skew: ...):
1. calculate difference between 2 futures prices (carry) and annualised it
(for commodities the back futures is ~1y away to avoid seasonality effect);
2. re-base the above (vol-adjusted annualised) carry by the mean of its
   absolute values over 1 year.


* Average the two signals (both capped at +/- 2) to get
  an 'avg_signal' for each futures.


* Calculate percentage position (weight) for each futures 'i' as
```

[9]Insert here that capital should be increased only by the square root of profits and decreased by (always the total amount of) losses.

```
25
26 weight_i = PortW_i * avg_signal_{i, t} * target_vol/vol_{i, t}
27
28 where
29 - PortW_i is the % allocation for futures i (and sum_i PortW_i = 1),
30   e.g. 1/ num_futures;
31 - vol_{i, t} is the annualised return st dev for futures price i at time t
32
33
34 one can add constraint such as
35 weight_i s.t. sum(abs(weights_i) = 1.5
36 ##########
37
38 NB: positions expressed as no. of futures (or lots) is simply then:
39 units_i =  weight_i * Capital_t/ (fut_{i,t} * mult_i * FX_{i,t})
40
41 where
42 - fut_{i,t} is the price for futures i at time t
43 - mult_i is the number of units embedded in the futures contract
44  (e.g. (a) 1,000 in regards of WTI futures, being the quoted price for
45  1 barrel or (b) 10 in respect of Eurostoxx50 idx futures, since each
46  index point worth 10 Euros).
47
48 Interpretation:
49
50 Capital_At_Risk        =    Capital_t * target_vol
51 LotValue_{i, t}        =    fut_{i,t} * mult_i * FX_{i,t}
52 LotValue_At_Risk{i, t} =    LotValue_{i, t} * vol_{i, t}
53
54 units_i = PortW_i * avg_signal_{i, t} * Capital / LotValue{i, t} * target_vol/vol_{i
55 = PortW_i * avg_signal_{i, t} * Capital_At_Risk / LotValue_At_Risk{i, t}
56 """
57 import numpy as np
58 import pandas as pd
59 from collections import deque
60
61 from quantopian.algorithm import order_optimal_portfolio
62 import quantopian.optimize as opt
63
64
65 def initialize(context):
66
67 # Dictionary of ContinuousFutures (sector in sub-dictionaries)
68 # where default -> continuous_future(root_symbol, offset=0, roll='volume', adjustment
69 context.futures_dict =  {
70 'equities' : {
71 'ES': continuous_future('ES'),    # S&P 500 e-Mini Futures; 'SP' standard
72 #         'ER': continuous_future('ER'),    # Russell 2000 e-Mini (US smalll cap)
73 #         'NQ': continuous_future('NQ'),    # Nasdaq e-Mini (US tech)
```

```
'NK': continuous_future('NK'),    # Nikkei 225 Futures (Japan)
#         'EI': continuous_future('EI'),    # MSCI Emerging Markets Mini (Emerging)
},
'fixedincome': {
#         'TU': continuous_future('TU'),    # 2yr Tbill
'TY': continuous_future('TY'),    # TNote 10 yr
'US': continuous_future('US'),    # TBond 30 yr
#         'ED': continuous_future('ED'),    # Eurodollar
},
'currencies' : {
'EC': continuous_future('EC'),    # Euro
'ME': continuous_future('ME'),    # Mexican Peso
'AD': continuous_future('AD'),    # Australian dollar
'BP': continuous_future('BP'),    # British pound
#         'JE': continuous_future('JE'),    # Japanese YEN
#         'SF': continuous_future('SF'),    # Swiss franc
},
'commodities' :{
'CL': continuous_future('CL'),    # Light Sweet Crude Oil
'GC': continuous_future('GC'),    # Gold
'SV': continuous_future('SV'),    # Silver
'NG': continuous_future('NG'),    # Natural gas
'CN': continuous_future('CN'),    # Corn
'WC': continuous_future('WC'),    # Wheat
'SY': continuous_future('SY'),    # Soybean
'SB': continuous_future('SB'),    # Sugar
'HG': continuous_future('HG'),    # Copper
}
}

# List of  futures.
context.my_futures = []
context.dq_carry={}    # dict of deque
for asset in context.futures_dict:    # .keys()
for future in context.futures_dict[asset].values():
context.my_futures.append(future)

context.dq_carry[future] = deque(maxlen=252)
context.dq_carry[future].append(1.0)

# equally-weighted portfolio allocation for each futures (see 'PortW_i' above)
context.weight =  1.0 / len(context.my_futures)    # to change to avoid commodities
context.no_fut = len(context.my_futures)

# vol target (xx% annual vol)
context.vol_targ =  1.    # was: 0.40

# Rebalance daily at market open
schedule_function(daily_rebalance, date_rules.every_day(), time_rules.market_open())
```

```
123
124 # Plot the number of long and short positions we have each day.
125 schedule_function(record_vars, date_rules.every_day(), time_rules.market_close())
126
127 def daily_rebalance(context, data):
128 """
129 Rebalance daily following price trends.
130 """
131 # get weight for each futures              NB: which = 1 (ratio), =2 (tanh), =3 (fnct
132 pos = position(context, data)        # or: position_alt(context, data, which=2)
133
134 # Get the current contract for each futures
135 contracts = data.current(context.my_futures, 'contract')
136
137 # Open orders
138 open_orders = get_open_orders()
139
140 weights = {}
141 for future in context.my_futures:
142 # get current futures contract
143 future_contract = contracts[future]
144
145 # If the future is not None,  tradeable and doesn't have pending orders
146 if future_contract is not None and \
147 data.can_trade(future_contract) and future_contract not in open_orders:
148 weights[future_contract] = context.weight * float(pos[future])
149
150 # re-based abs(weights) to sum 100% or use 'opt.MaxGrossExposure(1.0)' below
151 #    den = np.sum(np.absolute(weights.values()))
152 #    if den > 0:
153 #        weights = {k: v / den for k, v in weights.items()}
154
155 # safety feature
156 #   weights = {k: v if (not np.isnan(v)) else 0.0 for k, v in weights.items()}
157
158 # Order the futures we want to the target weights above
159 order_optimal_portfolio(
160 opt.TargetWeights(weights),
161 constraints=[ opt.MaxGrossExposure(1.5),],        # opt.MaxGrossExposure(1.5),
162 )
163
164
165 def position(context, data, fast_1=15, clip_v=2):
166 """
167 Common sense implementation
168 """
169 # get historical prices (1 year > slowest slow MA)
170 hist = data.history(context.my_futures, 'price', 252, '1d')
171
```

```
# Calculate vol as EMA standard deviation of daily returns (over 3 months)
volatility = hist.pct_change().ewm(span=63).std() # or, com=31, halflife=22
ann_vol = volatility * np.sqrt(252)

#    TREND-FOLLOWING

# calc moving-avg-crosses (fast/slow: 15/60, 30/120, 60/240)
fast_2 = fast_1 * 2; fast_3 = fast_2 * 2
MA_cross1 = hist.ewm(span=fast_1).mean() - hist.ewm(span=fast_1*4).mean()
MA_cross2 = hist.ewm(span=fast_2).mean() - hist.ewm(span=fast_2*4).mean()
MA_cross3 = hist.ewm(span=fast_3).mean() - hist.ewm(span=fast_3*4).mean()

# (MA_cross stack on top of each other, then group by dates and calc mean)
MA_cross_mean = pd.concat([MA_cross1, MA_cross2,MA_cross3]).groupby(level=0).mean()

# scale MA cross mean by price normal vol (i.e. price * annualised return vol)
raw_signal = MA_cross_mean / (ann_vol * hist)    # df

# re-based by mean of the absolute MA-cross values over the past year & mult
trend_signal = raw_signal.tail(1) / np.mean(np.absolute(raw_signal[:-1]))
# vect/vect

# alternatives:
#    i. trend_signal = np.tanh(raw_signal)
#   ii. trend_signal = raw_signal * np.exp(-raw_signal*raw_signal / 4)
#  iii. trend_signal.clip(-clip_v, clip_v)   <- chosen here
trend_signal = trend_signal.clip(-clip_v, clip_v)

#    ##################    #
#        CARRY
#
carry = {}

for asset in context.futures_dict.keys():    # .keys()

_back = 5 if (asset == 'commodities') else 1

for future in context.futures_dict[asset].values():

# get contract chains (prices and expiries)
chain = data.current_chain(future)

front_contract = chain[0]
front_ct_price = data.current(front_contract, 'price')

back_contract = chain[ min(_back, len(chain)-1) ]    # check control: capped at max
back_ct_price = data.current(back_contract, 'price')

ct_num_days = (back_contract.end_date - front_contract.end_date).days
```

```
220
221 # check for bad data first
222 if (np.isnan(front_ct_price) or np.isnan(back_ct_price) or
223 np.isnan(ct_num_days) or ct_num_days==0):
224 # neutral values:
225 _carry = 0.0
226 context.dq_carry[future].append(1.0)
227 log.info("Bad data for ct: %s on time %s"
228 %(str(front_contract.root_symbol), str(get_datetime())) )
229 else:
230 # if backwardation (downward sloping) -> long, else (contango) -> short
231 raw_carry = (1.0 - back_ct_price / front_ct_price) * 252.0/ct_num_days
232 _carry = raw_carry / float(ann_vol[future].tail(1))
233 context.dq_carry[future].append(_carry)
234
235 carry_base = 0.33 if (len( context.dq_carry[future])<50) else \
236 np.mean(np.absolute(context.dq_carry[future]))
237 carry[future] = _carry / carry_base
238
239 # combine the two signals (nb: carry converted from dict to df, for easy of calcs)
240 sum_signals = trend_signal.add(pd.DataFrame([carry], columns=carry.keys(), index=tre
241
242 # finally, scale the signal by the vol target (up to 4x)
243 return  ( sum_signals/2.0 * context.vol_targ/ann_vol.tail(1)).clip(-clip_v*4, clip_v
    # position
244
245 def record_vars(context, data):
246 """
247 This function is called at the end of each day and plots
248 the number of long and short positions we are holding.
249 """
250 # get no. of long and short positions
251 longs, shorts = 0, 0 # or longs = shorts = 0
252 for position in context.portfolio.positions.itervalues():
253 if position.amount > 0:    longs += 1
254 elif position.amount < 0:    shorts += 1
255
256 # Record no of long or short futures
257 record(long_count=longs, short_count=shorts)
258 #
259 """
260 # calculate diversification multiplier = 1/sqrt(w.T * Corr * w), capped at 2.5
261 #   hist_weekly = hist.resample('1W').last()    # use weekly data
262 #   correlation = hist_weekly.corr()
263 #   w = np.ones(context.no_fut) * 1.0/ context.no_fut
264 #   mult =  = min(2.5, 1.0/np.sqrt(np.dot(w.T, np.dot(correlation, w))) )
265 """
```

Chapter 2

Classical Strategies

blah blah blah

2.1 Momentum

This is main ingredients of CTAs . . . Explanations abound about the reasons for such effects that should be rather limited in an efficient market. Prof. Fama itself, a proponent of EMH, defined momentum as the greatest puzzle of finance and does take it into account for his funds (Dimensional Investors).

2.1.1 Probabilistic cross-momentum

We start from the simple strategies of allocating across markets (e.g. ETFs). The purposes of these is to be used in a pension wrapper where trading costs are high (e.g. in UK around 12.50 per trade) and instruments limited to cash-ones (we will be using ETFs).

As we mentioned, one of the issues of momentum is the risk of **whipsaw** (quick reversing of positions). The usual solution in academia (and even some sophisticated practitioners) is to calculated and implement the signal **periodically** on a monthly basis. A slightly more sophisticated version is to calculate the signal daily and implement it only if **statistically significant**. The obvious advantage is to be more reactive to changes.

But what we mean by statistically significant. One way is using the t-stat for the difference between the mean of two assets (statistics 101).

. . . TO BE COMPLETED . . .

In Python:

```
# get daily returns
self.price = self.History(self.symbols, self.back_period,
Resolution.Daily)["close"].unstack(level=0)      # .dropna(axis=1)
daily_rtrn = self.price.pct_change().dropna() # or: np.log(self.price / self.price.sl

_n = len(daily_rtrn.index); sqrt_n = np.sqrt(_n)

# reset
self.max_prob = -1.; target = None

# TODO: make it more Pythonic!
for i, tkr_1 in enumerate(self.symbols):
for j, tkr_2 in enumerate(self.symbols):

if i < j: # upper part matrix(tkr_1, tkr_2); (n^2 - n) / 2 operations

rtrn_diff = daily_rtrn[tkr_1.Value] - daily_rtrn[tkr_2.Value]

# t_stat = avg(diff)/(std(diff)/sqrt(n))
x = np.mean(rtrn_diff) / np.std(rtrn_diff) * np.sqrt(_n)

if x > 0:    # x>0 -> tkr_1 better than tkr_2
prob = (1 - t.sf(x, _n-1))  # T-dist cum prob: Prob(X<x)
if prob > self.max_prob:
self.max_prob = prob
target = tkr_1
else:        # x<0 -> tkr_2 better than tkr_1 (reverse)
prob = (1 - t.sf(-x, _n-1))                      # NB: -x
if prob > self.max_prob:
self.max_prob = prob
target = tkr_2   # NB: tkr_2

# check prob above min level
if self.max_prob > self.pr_lvl: self.target = target

if self.target is None: return
# check if already in
if self.Portfolio[self.target].Quantity != 0: return

# switch
self.Liquidate()
```

```
self.SetHoldings(self.target,1.)
```

2.1.2 Time-series momentum

. . . TO BE COMPLETED . . . also called trend-following or absolute momentum

```
"""
scipy.signal.lfilter(b, a, x, axis=-1, zi=None)

b : array_like
a : array_like (If a[0] is not 1, then both a and b are normalized by a[0])

x : array_like (N-dimensional input array).

The filter function is implemented as a direct II transposed structure. This means th
a[0]*y[n] = b[0]*x[n] + b[1]*x[n-1] + ... + b[M]*x[n-M] ...
- a[1]*y[n-1] - ... - a[N]*y[n-N]

The rational transfer function describing this filter in the z-transform domain is:
-1              -M
b[0] + b[1]z  + ... + b[M] z
Y(z) = -------------------------------- X(z)
-1              -N
a[0] + a[1]z  + ... + a[N] z

"""
import numpy as np
import pandas as pd

from scipy.signal import butter, lfilter, freqz, lfilter_zi
import matplotlib.pyplot as plt

from scipy.stats import linregress

from numpy import log, polyfit, sqrt, std, subtract, square
from numpy.random import randn


# TODO: create class: init (below) and step() (keeping a couple of past values)
def lowpass(data, period, zi=None):
"""
2-poles low-pass digital filter

var LowPass(var *Data, int Period)
{
var* LP = series(Data[0]);
```

```
var a = 2.0/(1+Period);
return LP[0] = (a-0.25*a*a)*Data[0]
+ 0.5*a*a*Data[1]
- (a-0.75*a*a)*Data[2]
+ 2*(1.-a)*LP[1] - (1.-a)*(1.-a)*LP[2];
}

"""
a = 2. / (1+period)
num_b = np.array( [a-0.25*a*a, 0.5*a*a, -(a-0.75*a*a)] )
den_a = np.array( [1.         , -2*(1.- a), (1.- a)*(1.- a) ] )
# NB: signs

if (not isinstance(data, np.ndarray)): data = np.array(data)
# just in case

# to start filter from first data point
if zi is None:
y = lfilter(num_b, den_a, data)
else:
_zi = lfilter_zi(num_b, den_a) * data[0]
y, _ = lfilter(num_b, den_a, data, zi=_zi)

return y

# TODO: check corner cases
def MMI(data, period):
""" My (vectorised) implementation of Zorro's Market Meanness Index
"""
if not isinstance(data, np.ndarray): data = np.array(data)  # just in case
period = int(min(period, len(data)-1, 1000))
if period < 2: return 75

# m = np.median(data) # for all data points
m = pd.Series(data).rolling(window=int(period), min_periods=1).median().values

nl = np.logical_and(data[1:] > m[1:], data[1:] > data[:-1])  # or a & b, np.logical_a
nh = np.logical_and(data[1:] < m[1:], data[1:] < data[:-1])

mmi_array = np.zeros(shape=data.shape[0]-1)  # get array of zero values; n-1
mmi_array[nl] = 1.
mmi_array[nh] = 1.

# # alternative: if data was a pd.DataFrame
# longs = df.close > df.close.shift(1); ...
# df['pos'] = 0; df.loc[longs,'pos'] = 1; df.loc[shorts,'pos'] = -1

ret = np.cumsum(mmi_array)  # , dtype=float)
ret[period:] = ret[period:] - ret[:-period]
```

```
return 100. * ret[period - 1:] / period
```

```
# loop

num_valid_quotes=0.; record_date_once = True; traded=False
# for each ticker
for _, tkr in enumerate(self.tickers):
# check data is available (missing/corrupt sometimes)
if not data.ContainsKey(tkr): continue

# record new incoming data
num_valid_quotes += 1
self.prices[tkr].append(float(data[tkr].Close)) # self.Portfolio[tkr].Price # self.S

# record date if at least one data
if num_valid_quotes>0. and record_date_once:
self.data_time.append(data.Time)  # .to_pydatetime()  # from TimeStamp to datetime
record_date_once = False

prices_array = np.array(self.prices[tkr])
weight = self.EW # * 0.5 * OptimalF * Capital * sqrt(1 + ProfitClosed/Capital)

# 1. TREND (every hour)
trend = lowpass(prices_array, self.hist_period)

peak, valley = peak_valley(trend)

# MMI_raw = MMI(prices_array, int(0.6*self.hist_period))  # 200 bars
# MMI_smoth = lowpass(MMI_raw, self.hist_period, "fit from start")

trending = regr_rvalue(self.x, prices_array[-self.x_range:]) > 0.5
# trending = hurst(prices_array, 200) > 0.5

if trending:  # True or trending or falling(MMI_smoth):
if valley:
self.SetHoldings(tkr, weight)
elif peak:
self.SetHoldings(tkr, -1. * weight)
```

2.2 Mean-reversion

Mean-reverting strategies are aplenty (easy to construct using two or more assets), exploitable on on different time frames (from seconds e.g. market making, to months for 'quantmental'), justifiable (at least vs. momentum) with a nice story behind (e.g. similar economies for country ETFs, or mining

vs. gold); yet, when they break up, the do suddenly and spectacularly (e.g. LTCM).

The 75 percent rule

[Show mathematically that IID mean-revert 75% of the time ... Generalise to non-IID case by locality prop....]

First, one should not confuse mean-reversion of returns (more laborious to trade) vs. mean-reversion of prices (i.e. *anti-serial-correlation* of returns), which we can easily trade. Moreover, there are 2 kind of strategies (i) **time-series** and (ii) **cross-sectional** mean reversion, the latter means that the cumulative rtrns of a basket of instruments will revert to their cumulative average, whilst the first (more traditional) means the prices are meant to revert to their own historical mean.

Few *price* series are found to be really mean-reverting and hence called *stationary*[1], but we can get more of these by combining two or more series together (*cointegrating* series - CADF test and Johansen test (see A.1), the latter gives us also weights of each asset to create a mean-reverting portfolio).

In words, **mean-reversion** means that next price series change is proportional to the difference between current price and mean price. ADF (Augmented Dickey-Fuller) test null hypothesis is, indeed, that the proportionality constant is zero:

$$\Delta y(t) = \lambda y(t-1) + \mu + \beta t + \alpha_1 \Delta y(t-1) + \ldots + +\alpha_k \Delta y(t-k) + \epsilon_t \quad (2.1)$$

The ADF test statistic is $\lambda / SE(\lambda)$ (for mean reversion we expected λ to be negative; the null hypothesis is obviously for $\lambda = 0$ and the test needs to be more negative than the critical values to reject the null hypothesis) - for daily series we usually assume the drift β to be around zero, the mean μ non-zero and the lag $k = 1$.

Stationarity means that the variance of the log of the prices increases

[1] Mean reversion and stationarity are equivalent ways of looking at the same phenomenon, but give rise to two different statistical, respectively ADF and variance ratio.

slowly (i.e. as a sub-linear function of time) than that of a geometric random walk (which is a linear function of time), such a function is approximated by $(\Delta t)^{2H}$ where H is the Hurst exponents: if $H < 0.5$ the series is stationary, if $= 0.5$ a geometric random walk, and above the series is trending - NB: it is enough for the variance to be sub-linear (i.e. $H < 0.5$, not necessary to be zero. i.e. not necessarily range bound when H is zero). An hypothesis test is provided by the Variance Ratio test: an

$$\frac{Var(z(t) - z(t-\tau)}{\tau\, Var(z(t) - z(t-1))}$$

where z is the log of the price and τ an arbitrary lag (see implementation below of Hurst for the numerator, denominator is the usual variance).

In Python (note the half-time of mean-reversion comes from analytical solution of just re-expressing the ADF equation above as an Ornstein-Uhlenbeck process):

```
import statsmodels.tsa.stattools as st
# import matplotlib.pylab as plt
import numpy as np
import pandas as pd

df_caus = pd.read_csv('USDCAD.csv')
plt.hold(False)
df_caus.y.plot()
#plt.show()
print st.adfuller(df_caus.y, maxlag=1)

import hurst as h
print h.hurst(df_caus.y)

from arch.unitroot import VarianceRatio
vr = VarianceRatio(np.log(df_caus.y))
print(vr.summary().as_text())
```

Where imported module for the Hurst component is:

```
# hurst.py
from numpy import cumsum, log, polyfit, sqrt, std, subtract, square
from numpy.random import randn

def hurst(data, bar_lags=100):
"""Returns the Hurst Exponent of the time series (data),
it helps test whether the time series is:
```

```
(i)   Trending (H > 0.5)
(ii)  Random Walk (H ~ 0.5)
(iii) Mean reverting (H < 0.5)
"""
if not isinstance(data, np.ndarray): data = np.array(data)

# Create the range of lag values
lags = range(2, bar_lags)

# Calculate the array of the variances of the lagged differences
tau = [sqrt(std(subtract(data[lag:], data[:-lag]))) for lag in lags]
# Use a linear fit to estimate the Hurst Exponent
poly = polyfit(log(lags), log(tau), 1)

# alternative (closer to usual formula): same result
# tau2  = [square(std(subtract(ts[lag:], ts[:-lag]))) for lag in lags]
# poly2 = polyfit(log(lags), log(tau2), 1)
# hurst = poly2[0] / 2.0

# Return the Hurst exponent from the polyfit output
return 2 * poly[0]
```

Another interpretation of λ is evident transforming ADF equation in a differential equation:

$$dy_t = (\lambda y_{t-1} + \mu)dt + d\epsilon_t \qquad \text{eq. dif}$$
$$E(y_t) = y_0 e^{\lambda t} - \mu/\lambda(1 - e^{\lambda t}) \qquad \text{solution}$$

which links the half-life of mean-reversion as equal to $-ln(2)/\lambda$ (recall that lambda is negative for a mean-reverting process). Since we really need some mean-reversion to be profitable, the half-line tends to be more useful in practice than the quite restrictive ADF or variance ratio tests: it gives us an indication of the lookback and expected holding period for our trading strategy, e.g. if the half-line is one month we cannot really implement mean-reversion on smaller frequencies (better not pursuing further since there will not be many opportunities).

In Python, the half-line is:

```
import statsmodels.formula.api as smf

def halflife(df,col):
df['ylag'] = df[col].shift(1)
df['deltaY'] = df[col] - df['ylag']
```

```
results = smf.ols('deltaY ~ ylag', data=df).fit()
lam = results.params['ylag']
halflife=-np.log(2)/lam
return lam, halflife
```

Important to run mean-reversion tests because we make use of all (daily) data, rather than just tests the strategy that rests only on a handful of trades.

2.2.1 One asset linear mean-reverting algo

Once we determine that prices are (sort of) mean-reverting and the half-life not (too) long, we can device a simple strategy: no. of units to buy/sell are to be proportional to the negative of normalised price deviation (i.e., ratio of (i) price less its moving average to (ii) rolling std deviation of prices[2]) using half-life as lookback period. Note the apparent theoretical contradiction that mean and variance should be constant, but we update them.

Cointegration

Cointegrated series are two or more series whose linear combination is stationary. Most common is pairs-trading: long one asset vs. short another.

There are two classical tests: Co-integrated Augmented Dickey-Fuller (or **CADF**, which works just for pairs) or Johansen (which works for more that two series). In the first test (CADF) a linear regression is first run to determine the hedge ratio between the two assets (important: use each as independent and select the one with t-stat more negative[3]) and then perform a stationary test run on the combination of the two. The second is a matrix generalisation of ADF (eigenvector decomposition of lambda matrix)

Example of CADF application (pair test): two ETFs for commodity countries (e.g Canada vs. Australia). Test is strongly dependent on the hedge ratio (pre-determined for example via linear regression) that follows

[2] You can think of this ratio as a z-score.

[3] Johansen tests may give two cointegrating relationships (2 hedge ratios) even for two price series, since we don't need to run the regression twice as in CADF

the choice of dependent/independent variables (use each variable as independent and select the one with best, ie. more negative, t-stat - since hedge ratio is order dependent). For pairs one can also check cointegration using the standard ADF on the portfolio of pair combined via the hedge ratio.

```
# import matplotlib.pylab as plt
import pyconometrics
import pandas as pd
import statsmodels.tsa.stattools as st
import hurst

# Canada and Australia (commodity based economies)
df = pd.read_csv('ETF.csv',index_col=0)
df[['ewa','ewc']].plot()

# scattering along a line
plt.scatter(df['ewa'],df['ewc'])
plt.title('ewa / ewc')
#plt.show()

# 1. one should really use each variable as independent
# and select the best (most negative) t-stat
import statsmodels.formula.api as smf
results = smf.ols('ewc ~ ewa', data=df).fit()
hedgeRatio = results.params['ewa']

# 2. form portfolio
df['coint'] = df['ewc']-hedgeRatio*df['ewa']

# 3. check cointegrated
print(pyconometrics.cadf(np.matrix(df['ewa']).H,
np.matrix(df['ewc']).H, 0, 1))
print('hurst', hurst.hurst(df['coint']))
print(st.adfuller(df['coint'],maxlag=1))
```

Johansen test generalises the ADF equation (y variables become vectors and λ a matrix): the proportionality matrix (λ) has a rank of r, which is calculated in two different ways: trace statistic and eigen statistic (both based on eigenvector decomposition). Null hypothesis (no cointegration) is that:

$$r = 1, r <= 1, \ldots, r <= n - 1$$

and if all rejected then $r = n$.

2.2.2 Multi-asset linear mean-reverting algo

This is the multi-assets version of strategy 2.2.1:

- check no. of cointegrating relationships (both Johansen tests), e.g. 3 for 3 ETFs;
- use strongest eigenvector (out of 3) as hedge ratios ad form a portfolio;
- find half-life of portfolio (earlier was of a single asset);
- similarly to the single asset case, get no. of units proportional (hence the name 'linear' for the strategy) to the negative of the normalised price (z-score); lookback for mean and st dev is the half life above;
- multiply hedge ratios (to get 'unit' portfolio) by no. of units to get each asset position.

This linear mean-reverting strategy is obviously not a practical strategy, at least in its simplest version (i.e. constant infinitesimal rebalancing and unlimited capital). Yet it is Sharpe is around 1.4 (with no transaction costs - [TO BE DONE: add]) and CAGR of 12%.


```
import statsmodels.tsa.stattools as st
import numpy as np
import pandas as pd

from johansen import coint_johansen

#1 check both tests: e.g. 3 cointegrating relationships
cols = ['ewc','ewa','ige']
res3 = coint_johansen(df[cols], p=0, k=1)

#2 se strongest eigenvector to form a portfolio
# NB:  eigenvectors (".evec") are rows (vs columns in Matlab)
df['yport'] = np.dot(df[cols], res3.evec[:,0])

# 3 find, similarly to before, the half life of the portfolios
import halflife
hf = halflife.halflife(df, 'yport')[1]

# 4. units proportional to negative of z-score (as earlier)
data_mean = pd.rolling_mean(df['yport'], window=hf)
data_std = pd.rolling_std(df['yport'], window=hf)
df['numUnits'] = -1*(df['yport']-data_mean) / data_std

# 5. get positions
```

```
tmp1 = np.ones(df[cols].shape) * np.array([df['numUnits']]).T
tmp2 = np.ones(df[cols].shape) * np.array([res3.evec[:,0]])
positions = tmp1 * tmp2 * df[cols]
positions = pd.DataFrame(positions)

# calculate pnl and equity line
pnl = positions.shift(1) * (df[cols] - df[cols].shift(1))  / df[cols].shift(1)
pnl = pnl.sum(axis=1)  # across columns: portfolio rtrb

ret = pnl / np.sum(np.abs(positions.shift(1)),axis=1)

plt.plot(np.cumprod(1+ret)-1)
plt.show()

print('APR', ((np.prod(1.+ret))**(252./len(ret)))-1) # =12%
print('Sharpe', np.sqrt(252.)*np.mean(ret)/np.std(ret))  # =1.37
```

Up to know we discussed about mean reversion of **prices** - Δy in equation (2.1) is the difference between two consecutive (e.g. daily) price and we constructed portfolios where hedge ratios were either number of shares or percentage of wealth $y_1 = h_1 y_1 + h_2 y_2 + \ldots$. What if the **log of prices** are co-integrated? That is possible, but more difficult to trade: the ADF equation becomes $\Delta \ln(y) = h_1 \Delta \ln(y_1) + \ldots$ that is approximately equal to the returns (i.e. $\Delta y / y = \Delta y_1 / y_1 \ldots$ - just recall Taylor series for logarithm) and therefore it implies a lot of rebalancing because of th changing price in the denominator.

Some traders like to use the **ratio** between two prices. Except for the case of two prices series with one hedge ratios being the opposite of the other (and hence $log(y_1/y_2)$ or y_1/y_2 are indeed stationary), a ratio of two prices is not necessarily a stationary series (but it can still be useful in some cases - for example, in cross-currencies, we may trade two pairs to get a less liquid one - even if pnl are in different currencies)

Mean reversion for Stocks and ETFs

. . . TO BE COMPLETED . . .

Mean reversion for Currencies

. . . TO BE COMPLETED . . .

Mean reversion for Futures

. . . TO BE COMPLETED . . .

2.2.3 Cross-sectional mean reversion

Passive aggressive mean reversion (PAMR) strategy for portfolio selection. There are three variants with different parameters, see original article for details (B. Li, P. Zhao, S. C.H. Hoi, and V. Gopalkrishnan, Passive aggressive mean reversion strategy for portfolio selection, 2012)

Financial explanation:


```
x_tilde[i] = data[stock].price
# or x_tilde[i] = data[stock].mavg(5) / data[stock].price

x_bar = x_tilde.mean()

# market relative deviation
mark_rel_dev = x_tilde - x_bar

# Expected return with current portfolio
exp_return = np.dot(context.b_t, x_tilde)

weight = context.eps - exp_return

variability = (np.linalg.norm(mark_rel_dev))**2

if variability == 0.0: step_size = 0 # no portolio update
else:  step_size = max(0, weight/variability)

b = context.b_t + step_size * mark_rel_dev
b_norm = simplex_projection(b)
```

```
context.b_t = b_norm
```

Python implementation (adapted from: INSERT REF . . . TO BE COMPLETED . . .)

```
#   #   #   #
# initialiation (just once)
self.pamr = PAMR(eps=0.0005, C=500, variant=0)  # <-- params
self.previous_w = self.pamr.init_weights(len(self.tickers)) # EW weights

# get data at every step
self.data_hist[tkr].append(float(data[tkr].Close))
self.data_time.append(data.Time)  # .to_pydatetime()  # from TimeStamp to datetime
df_price = pd.DataFrame(self.data_hist, columns=self.data_hist.keys(),
index = self.data_time)

rel_prices = (df_price / df_price.shift(1)).dropna()
last_rel_prices = np.asarray(rel_prices.tail(1)).flatten()  # work-around returns.il

# get weights
weight = self.pamr.step(last_rel_prices, self.previous_w)
self.previous_w = weight  # recorded it for next step evaluation
#   #   #   #   #   #   #
#       PAMR class
class PAMR():

def __init__(self, eps=0.5, C=500, variant=0):
"""
:param eps: Control parameter for variant 0. Must be >=0, recommended between 0.5 and
(se usi 'returns' - invece di ratios - allora eps <=0)
:param C: Control parameter for variant 1 and 2. Recommended value is 500.
:param variant: Variants 0, 1, 2 are available.
"""

# input check
if not(eps >= 0): raise ValueError('epsilon parameter must be >=0')

if variant == 0:
if eps is None: raise ValueError('eps parameter required')
elif variant == 1 or variant == 2:
if C is None: raise ValueError('C parameter require')
else:
raise ValueError('variant is a number from 0,1,2')

self.eps = eps
self.C = C
self.variant = variant


```

```
def init_weights(self, m):
return np.ones(m) / m


def step(self, x, last_b):
# calculate return prediction
b = self.update(last_b, x, self.eps, self.C)
return b

def update(self, b, x, eps, C):
""" Update portfolio weights to satisfy constraint b * x <= eps
and minimize distance to previous weights. """
x_mean = np.mean(x)
le = max(0., np.dot(b, x) - eps)

if self.variant == 0:
lam = le / np.linalg.norm(x - x_mean)**2
elif self.variant == 1:
lam = min(C, le / np.linalg.norm(x - x_mean)**2)
elif self.variant == 2:
lam = le / (np.linalg.norm(x - x_mean)**2 + 0.5 / C)

# limit lambda to avoid numerical problems
lam = min(100000, lam)

# update portfolio
b = b - lam * (x - x_mean)

# project it onto simplex
return simplex_projection(b)


def simplex_projection(v, b=1):
""" Projection vectors to the simplex domain

Implemented according to the paper: Efficient projections onto the
l1-ball for learning in high dimensions, John Duchi, et al. ICML 2008.

Optimization Problem: min_{w}\| w - v \|_{2}^{2}
s.t. sum_{i=1}^{m}=z, w_{i}\geq 0
Input: A vector v \in R^{m}, and a scalar z > 0 (default=1)
Output: Projection vector w

:Example:
>>> proj = simplex_projection([.4 ,.3, -.4, .5])
>>> print proj
array([ 0.33333333, 0.23333333, 0. , 0.43333333])
>>> print proj.sum()
1.0
```

```

Original matlab implementation: John Duchi (jduchi@cs.berkeley.edu)
"""

p = len(v)

# Sort v into u in descending order
v = (v > 0) * v
u = np.sort(v)[::-1]     #    = sorted(v, reverse=True)
sv = np.cumsum(u)

rho = np.where(u > (sv - b) / np.arange(1, p+1))[0][-1]
theta = np.max([0, (sv[rho] - b) / (rho+1)])
w = (v - theta)
w[w<0] = 0.

return w
```

2.2.4 Stat Arb

?? A [starting] Statistical Arbitrage strategy:

1. Select shares universe (e.g. i. a static list or ii. a dynamic one, like most liquid shares in same sector) and get historical prices
2. Find risk factors (i.e. common drivers for the returns), using e.g. PCA, ICA, etc.
3. Regress returns vs. risk factors to get individual factor exposure (betas)
4. find shares weights such that:
   - maximise your alpha (e.g. z-score of regression residuals * weights)
   - subject to some constraints, e.g. zero net exposure, gross exposure ¡= 100%, neutralised betas


```
# get log returns for the universe selection (NB: hist should be a pandas Panel)
symbols = [x.Value for x in self.fine_symbols] # x.Symbol.Value
for tkr in symbols:
self.AddEquity(tkr, Resolution.Daily) # NB: actually needed

hist = self.History(symbols, self.back_period, Resolution.Daily)
if hist is None: return
```

```
prices = hist["close"].unstack(level=0).dropna(axis=1)

logRtrn = np.log(prices / prices.shift(1)).dropna() # (num_rtrn, symbols)

# model: #1 risk factor
factors = PCA(1).fit_transform(logRtrn)    # fit + transform # (num_rtrn, [1])

X_c = smapi.add_constant(factors) #  factor(s) + const
model = smapi.OLS(logRtrn, X_c).fit()
betas = np.asarray(model.params.T)[:, 1:]   # model.params is (2, symbols) array; re

# model.resid is (days, symbols) array :=>
#   sum across days (.sum(axis=0)) for two periods [-x:,:],
#   standardise resulting 1-d array across symbols (by zscore) and
#   subtract older score from more recent
signal = sp.stats.zscore(np.asarray(model.resid)[-2:, :].sum(axis=0)) \
- sp.stats.zscore(np.asarray(model.resid)[-20:, :].sum(axis=0))

# optimised weights
w = self.get_weights(signal, betas)

# w_i re-based to 100% gross
denom = np.sum(np.abs(w))
if denom == 0: denom = 1.
w = w / denom

# remove tkrs we don't have (NB: prices.columns == self.fine_symbols)
for tkr in self.Portfolio.Values:
if (tkr.Invested) and  (tkr not in self.fine_symbols):
self.Liquidate(tkr.Symbol)

# submit orders
for i, tkr in enumerate(prices.columns):
self.SetHoldings(tkr, w[i])

self.trade_flag = False

def get_weights(self, signal, betas):
(m, n) = betas.shape    # (symbols, 1) if PCA(1)
x = cvx.Variable(m)
objective = cvx.Maximize(signal.T * x)

constraints = [cvx.abs(sum(x)) < 0.001,     # net ~= 0
sum(cvx.abs(x)) <= 1,         # gross <= 100%
x <= 3.5 / m, -x <= 3.5 / m] # weights upper/lower bounds

# neutralise all risk factors exposures (here just #1): abs(beta * weights) = 0
for i in range(0, n):
constraints.append(cvx.abs(betas[:, i].T * x) < 0.001)
```

```
57
58 probl = cvx.Problem(objective, constraints)
59 probl.solve(solver=cvx.CVXOPT)
60
61 # if no solution
62 if probl.status != 'optimal':
63 print(prob.status)
64 return np.asarray([0.0] * m) # return 0. array
65
66 return np.asarray(x.value).flatten()
```

2.3 Carry

2.3.1 Equity volatility

[UPDATE: use vol-adapted RSI as momentum indicator on bi-hourly prices, rather than simple MA]

In a nutshell, the algo either:

1. goes short vol (i.e. long XIV): if both i. VIX futures in contango (VXV > VIX) & ii. VIX is declining (not just when high enough like in the previous algos), or
2. goes long vol (i.e. long VXX): if both i. VIX futures in backwardation (VXV < VIX) & ii. VIX is increasing.
3. Otherwise, the algo take all position off (flat).


```
1 #....
2 # xiv_current = data.current(context.XIV, 'price')
3 # xiv_mean = data.history(context.XIV, "price", 60, "1d").mean()
4 vix_moma = 1 - (pd.Series(vix).rolling(window=60).mean()[-1]
5 / context.vix)
6
7 # params
8 r1 = 0.10
9 r2 = 0.15
10
11 if data.can_trade(context.XIV):
12 # short vol (i.e. long XIV)
13 if ( (context.vxv / context.vix > (1+r1)) & (vix_moma < r2)):
14 order_target_percent(context.XIV, 1)
15 order_target_percent(context.VXX, 0)
16 # long vol (i.e. long VXX)
```

```
elif ((context.vxv / context.vix < (1-r1)) & (vix_moma > r2)):
order_target_percent(context.XIV, 0)
order_target_percent(context.VXX, 1)
# flat (no position)
else:
order_target_percent(context.XIV, 0)
order_target_percent(context.VXX, 0)
#
```

```
# calc [MODEL #2] signal (-1: short vol, +1: long vol, 0: none)
self.RSI_previous = self.RSI_previous_s if self.use_shorter_RSI else self.RSI_previou
RSI_curr = self._RSI_s.Current.Value if self.use_shorter_RSI else self._RSI_l.Current

# short vol
if self.RSI_previous > 85 and RSI_curr <= 85: # down and below 85: none
self.m2_sign = 0.0
if self.RSI_previous < 70 and RSI_curr >= 70: # up and above 70: short vol
self.m2_sign = -1.0
# long vol
if self.RSI_previous > 30 and RSI_curr <= 30: # down and below 30: long vol
self.m2_sign = +1.0
if self.RSI_previous < 15 and RSI_curr >= 15: # up and above 15: SELL
dfself.m2_sign = -1.0
```


2.3.2 Currencies

[LIST OF ARGS] Carry trade: the UIP and CIP, macro-economic forecasts vs. reality, premiun for what

2.3.3 Rates

The oldest strategy - rumored to have been created by Salomon Bros in the early 1980's is the forward rate bias strategy on Eurodollar futures ...If one thinks about it is the bank business in its pure form: if one buys the 4th quarterly contract, it is like borrowing at 12 months and lending at 15 months.

Typically the Sharpe ratio of the strategy has been around 1 (returns are 1% p.a., so leverage is needed unless the strategy is used for liability trades[4]).

[4]These trades are beyond the scope of the present article. It suffices to say that derivatives (e.g. call options on aggregate performance) were sometimes embedded in

Credit

[...]

loans to corporates with the aim to cheapen the funding cost from the 2nd/3rd year onwards if strategy performance was positive.

Chapter 3

Machine learning

. . . TO BE COMPLETED . . .

Use some of my notes https://github.com/alex-muci/machine-learning-notes

In early 2000 it was SVM, five years later random forests were all the rage . . .

3.1 Ensembles

You get some simple rules and combine them to get a more complex one (learn over a subset of data that generates a rule, repeat, and combine all). This is very convenient in finance where overfitting (rather than underfitting) is the major issue.

Bagging

Bagging (Bootstrap AGGregatING) or Bootstrapping aggregation: combine simple rule(s) (e.g. 3rd order polynomial) on **random subsets** of the training data and average the result.

Bagging is the simplest ensemble learning we can think of:

- Simplest way to choose data: select uniformly randomly subsets (with *replacement*[1]).

[1]Replacement is needed because you are trying to create sample distributions that

- Simplest combination: simple average.

Such an ensemble learning tends to perform better on the test/validation datasets (lower variance) than more complex models (at the expense of an higher bias).

Note that Random Forests are related to Bagging (averaging across trees to reduce variance), but with an extra randomness added by sub-sampling features.

Boosting

Boosting improves Bagging by (i) focusing on the **'hardest'** examples and using (ii) **weighted** mean. One introduces a new error definition (rather than simple mismatch) as probability - given a distribution - that my Hp disagrees with the truth concept on some instances x : $Pr_{distr}[Hp(x) \neq c(x)]$.

... Boosting may overfit if (i) the underlying learner overfits from the start (e.g. ANN with many layers) any looping does not make it better, (ii) 'pink noise' (uniform noise). The second possibility make Boosting not a great choice in finance where noise-to-signal is very high, better to stick with Bagging ...

3.1.1 Random forests

Currently in the industry, random forests are usually preferred over SVM's.
... preferred traditional machine learning algo

Note that Random Forests ('**RF**') are related to Bagging (averaging across trees to reduce variance), but with an extra randomness added in the selection of the features. Suppose that there is *one very strong feature* in the data set. When using bagged trees, most of the trees will use that feature as the top split, resulting in an ensemble of similar trees that are *highly*

could have come for the original population; it introduces slightly more variation, which is good for generalisation.

correlated: averaging highly correlated quantities does not significantly reduce variance; by randomly leaving out candidate features from each split, **Random Forests "decorrelates" the tree**s, such that the averaging process can reduce the variance of the resulting model. In addition, to reduce potential overfit, one can increase the min_samples_split (a param in DecisionTreeClassifier class) from default of 2 (won't split samples lower than 2) to something bigger, e.g. 50.

INSERT CODE ...

3.2 Feature extraction or transformation

An important task when performing supervised learning is determining which features provide the most predictive power. The easiest (in-sample) solution is using a scikit-learn supervised learning algorithm that has a feature_importance_ attribute available for it (e.g. ensemble like RandomForestCLassifier or Adaboost). This attribute is a function that ranks the importance of each feature when making predictions based on the chosen algorithm.


```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier().fit(X_train, y_train)
# Extract feature importances
importances = model.feature_importances_

# plot
feature_plot(importances, X_train, y_train)

def feature_plot(importances, X_train, y_train):
    # Display the five most important features
    indices = np.argsort(importances)[::-1]
    columns = X_train.columns.values[indices[:5]]
    values = importances[indices][:5]

    # Creat the plot
    fig = pl.figure(figsize = (9,5))
    pl.title("Normalized Weights for First Five Most \
    Predictive Features", fontsize = 16)
    pl.bar(np.arange(5), values, width = 0.6, align="center", \
           color = '#00A000', label = "Feature Weight")
    pl.bar(np.arange(5) - 0.3, np.cumsum(values), width = 0.2, \
```

```
        align = "center", color = '#00A0A0', \p
        label = "Cumulative Feature Weight")
    pl.xticks(np.arange(5), columns)
    pl.xlim((-0.5, 4.5))
    pl.ylabel("Weight", fontsize = 12)
    pl.xlabel("Feature", fontsize = 12)

    pl.legend(loc = 'upper center')
    pl.tight_layout()
    pl.show()
```

Similar to feature selection, we can use different feature extraction / transformation techniques to reduce the number of features in a dataset (e.g. PCA, ICA, RCA, or linear combinations). The difference between feature selection and feature extraction / transformation is that while we maintain the original features with the first (e.g. when using 'sequential backward selection'), for the latter we transform or project the data onto a new feature space. In the context of dimensionality reduction, feature extraction is an approach to data compression with the goal of maintaining most of the relevant information whilst improving storage space or the computational efficiency of the learning algorithm, but - in our case - it is more to improve the predictive performance by reducing the curse of dimensionality (for example PCA is a general algo for feature extraction or transformation).
See example of PCA in Satistical Arbitrage, sub-section XXX

3.3 Online learning

3.3.1 Pattern-matching

. . . TO BE COMPLETED . . .

Pattern matching algos were developed in the last 10-12 years - hence, they are not to be confusing with technical analysis charts of head and shoulders or similar. They essentially consist of 2 steps:

- given a forecasting window, say 5 days, find out 'similar' windows in the past. Similarity is defined according to some quant metrics, e.g.

distribution, correlation (the CORN implemented below), KNN (TO BE ADDED), etc.

- optimise weights (vs. best performance or best sharpe ratio or ...) on similar past history only.

The gist of it all is that you find similar history and then optimised your weights on such relevant history. All these algos have the mathematical property of *universality*: in the long-term, they are optimal strategies.

TO DO: implement KNN and compare vs. authors' backtesting (even if they assume no transaction costs, ...)

```
# initialisation
self.model = CORN(window=self.window, rho=0.1)
self.previous_w = self.model.init_weights(len(self.tickers))

rel_prices = (df_price / df_price.shift(1)).dropna()
last_rel_prices = np.asarray(rel_prices.tail(1)).flatten()
weight = self.model.step(last_rel_prices, self.previous_w, rel_prices.iloc[-self.hist

# at each loop
weight = self.model.step(last_rel_prices, self.previous_w, rel_prices.iloc[-self.hist


class CORN():
  """
  Correlation-driven nonparametric learning approach. Similar to anticor but instead
  of distance of return vectors they use correlation.
  In appendix of the article, universal property is proven.
  Reference: B. Li, S. C. H. Hoi, and V. Gopalkrishnan. Corn: correlation-driven nonp
  """

  def __init__(self, window=5, rho=0.1, fast_version=True, min_history=0):
    """
    :param window: Window parameter.
    :param rho: Correlation coefficient threshold. Recommended is 0.
    :param fast_version: If true, use fast version which provides around 2x speedup,
     more memory.
    """
    # input check
    if not(-1 <= rho <= 1):
        raise ValueError('rho must be between -1 and 1')
    if not(window >= 2):
        raise ValueError('window must be greater than 2')

    self.window = window
    self.rho = rho
```

```
36      self.fast_version = fast_version
37      self.min_history = min_history
38
39      def init_weights(self, m):
40          return np.ones(m) / m
41
42
43      def step(self, x, last_b, history):
44          if len(history) <= self.window:
45              return last_b
46          else:
47              # init
48              window = self.window
49              indices = []
50              m = len(x)
51
52              # calculate correlation with predecessors
53              X_t = history.iloc[-window:].values.flatten()
54              for i in range(window, len(history)):
55                  X_i = history.iloc[i-window:i].values.flatten()
# was: X_i = history.iloc[i-window:i-1].values.flatten()
56                  if np.corrcoef(X_t, X_i)[0,1] >= self.rho:
57                      indices.append(i)
58
59              # calculate optimal portfolio
60              C = history.iloc[indices, :]
61
62              if C.shape[0] == 0: b = np.ones(m) / float(m)
63              else: b = optimal_weights(C)
64
65              return b
66
67      def optimal_weights(X, metric='return',
68          max_leverage=1, rf_rate=0., alpha=0., freq=252, no_cash=False, sd_factor=1.,
69          """ Find best constant rebalanced portfolio with regards to some metric.
70          :param X: Prices in ratios.
71          :param metric: what performance metric to optimize, can be either `return` or
72          :max_leverage: maximum leverage
73          :rf_rate: risk-free rate for `sharpe`, can be used to make it more aggressive
74          :alpha: regularization parameter for volatility in sharpe
75          :freq: frequency for sharpe (default 252 for daily data)
76          :no_cash: if True, we can't keep cash (that is sum of weights == max_leverage
77          """
78          assert metric in ('return', 'sharpe', 'drawdown')
79
80          x_0 = max_leverage * np.ones(X.shape[1]) / float(X.shape[1])
# max_leverage * np.array(X.shape[1] * [1. / float(X.shape[1])])
81
82          if metric == 'return':
```

```
83              objective = lambda b: -np.sum(np.log(np.maximum(np.dot(X - 1, b) + 1, 0.0
84          elif metric == 'sharpe':
85              objective = lambda b: -sharpe(np.log(np.maximum(np.dot(X - 1, b) + 1, 0.0
86              rf_rate=rf_rate, alpha=alpha, freq=freq, sd_factor=sd_factor)
87          elif metric == 'drawdown':
88
89          def objective(b):
90              R = np.dot(X - 1, b) + 1
91              L = np.cumprod(R)
92              dd = max(1 - L / np.maximum.accumulate(L))    # NICE!
93              annual_ret = np.mean(R) ** freq - 1
94              return -annual_ret / (dd + alpha)
95
96          if no_cash: # this is usually the one I see in finance
97              cons = ({'type': 'eq', 'fun': lambda b: max_leverage - sum(b)},)
98          else:
99              cons = ({'type': 'ineq', 'fun': lambda b: max_leverage - sum(b)},)
100
101         while True:
102             # problem optimization
103             res = optimize.minimize(objective,
104                 x_0,   # initial values: np.array(self.n * [1. / self.n])
105                 bounds=[(0., max_leverage)]*len(x_0),
106                 constraints=cons,
107                 method='slsqp', **kwargs)
108
109             # result can be out-of-bounds -> try it again
110             EPS = 1E-7
111             if (res.x < 0. - EPS).any() or (res.x > max_leverage + EPS).any():
112                 X = X + np.random.randn(1)[0] * 1E-5
113                 logging.debug('Optimal weights not found, trying again...')
114                 continue
115             elif res.success:
116                 break
117             else:
118                 if np.isnan(res.x).any():
119                     logging.warning('Solution does not exist, use zero weights.')
120                     res.x = np.zeros(X.shape[1])
121                 else:
122                     logging.warning('Converged, but not successfully.')
123                     break
124
125             return res.x  # res['x']  # opt_weights
```

3.4 Deep learning

[Start first by subsections on perceptrons XXX and Softmax XXX, in particular perceptron and gradient descent algorithms[2]].

Now were are in a position to put all the building blocks together and start from Neural Networks.

Neural Networks intro

To start building neural networks ('NN'), we combine two perceptrons (whose outputs are *linear* boundaries) into a third model that gets us *non-linear* boundaries - e.g. summing the two probabilities generated by the two perceptrons and use the sigmoid function to turn the such a sum into a probability (we can also use weights to assign more importance to one perceptron over the other).

We can enrich the NN architecture further by either:

- adding more **nodes** to the (i) input layer (very first layer: space dimensions), (ii) hidden layer (the set of linear models created by the inputs), and (iii) output layer (which combines the previous linear models to get a non-linear one - if it has a single node - or a *multi-class classification* if more nodes are added) - see fig XXX) - and/or
- add more **layers**, which is where the 'deep' NN magic happens.

INSERT FIGURE

In other words, neural networks are a bunch of perceptrons added together in parallel to form one layer and multiplying such layers separated by non-linear elements.

To recap: neural network can represent booleans, continuous or arbitrary functions by just adding more **units** (more perceptrons, e.g. visually organised in a column) and (hidden) **layers** (number of columns) - so no restriction bias. To avoid overfit than limit numbers of units/layers and use cross-validation (same the trade-off bias vs. variance in other methods.)

[2] A well written, free, extra resource for more details is deep learning book.

Feedforward is the process neural networks use to turn the inputs into an output: it takes the input vector and apply a sequence of linear models (vector and matrix multiplication) and sigmoid functions to return predictions (see fig. XXX)
INSERT FIGURE Just as before, neural networks will produce an error function, which at the end, is what we'll be minimizing to train a NN.

Backpropagation: get output given inputs by changing intermediate units and layers using gradient descent (one at the time).

In a nutshell, backpropagation consists of:

- Running the feedforward operation.
- Comparing the output of the model with the desired output and calculating the error.
- Running the feedforward operation *backwards* (backpropagation) to spread the error to each of the weights.
- Use this to update the weights, and get a better model.
- Repeat this until we have a model that is good.

We already did the math for the single perceptron case ...; for the multi-layer perceptron the math is similar but a bit more complicated:

- prediction: $\hat{y} = \sigma W^{(3)} \circ \sigma W^{(2)} \circ \sigma \circ W^{(1)}(x)$, a composition of functions;
- error function: similar, but with more complicated predictions;
- gradient of the error function: similar, but longer vector.

INSERT FIGURE

[Preference bias: algorithm's selection of one representation over another. Occam's razor.]

Keras

Keras is a high-level neural networks API, written in Python, and capable of running on top of TensorFlow (preferable), (Microsoft) CNTK, or Theano.

Unless for research purposes or developing special neural networks, Keras is the better choice since you can quickly build even very complex models. If more control over the network or debugging / watch closely is required,

TensorFlow ('TF') is the right choice (though the syntax can be sometimes a nightmare)[3].

The general idea for this example is that you'll first load the data, then define the network, and then finally train the network.

```
import numpy as np
from keras.utils import np_utils
import tensorflow as tf
# Using TensorFlow 1.0.0; use tf.python_io in later versions
tf.python.control_flow_ops = tf

# Set random seed
np.random.seed(42)

# a wrapper that treats the network as a sequence of layers
# common methods are compile(), fit(), and evaluate()
from keras.models import Sequential
from keras.layers.core import Dense, Activation

# X has shape (num_rows, num_cols), where the training data stored
# as row vectors
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)

# y must have an output vector for each input vector
y = np.array([[0], [0], [0], [1]]).astype('float32')

# One-hot encoding the output
y = np_utils.to_categorical(y) # , num_classes=1)
# or no need for the above and use 'binary_crossentropy'

# Create the Sequential model
model = Sequential()

# 1st Layer - Add an input layer of 32 nodes with same input shape
# as the training samples in X
model.add(Dense(32, input_dim=X.shape[1]))

# Add a softmax activation layer
model.add(Activation('softmax'))  # or 'tanh', ...

# two steps above are equivalent to:
# model.add(Dense(32, activation="softmax")))
# but common to separate the two for direct access

# 2nd Layer - Add a fully connected output layer (dim=1)
```

[3]Since Keras is going to be integrated in TF, it is likely wiser to build your network using tf.contrib.Keras and insert anything you want in the network using pure TensorFlow

```
model.add(Dense(1)) # since only two classes: 0 and 1

# Add a sigmoid activation layer
model.add(Activation('sigmoid'))

# call backend and binds loss fnc, optimiser and metrics
model.compile(loss="categorical_crossentropy",
optimizer="adam", metrics = ["accuracy"])
# see resulting model architecture
model.summary()

# train the model
model.fit(X, y, nb_epoch=1000, verbose=0)
# nb: in Keras 2 'nb_epoch' becomes 'epochs'

# if train and test, and batch_size
model.fit(x_train, y_train, epochs=10,
batch_size=32,  verbose=2,
validation_data=(x_test, y_test))

# finally see how the model does
score = model.evaluate(X, y)
print('Test loss:', score[0])
print("\nAccuracy: ", score[-1])

# Checking the predictions
print("\nPredictions:")
print(model.predict_proba(X))
```

Keras requires the input shape to be specified in the first layer, but it will automatically infer the shape of all other layers. This means you only have to explicitly set the input dimensions for the first layer.

Once we have our model built, we need to compile it before it can be run on any input data. Compiling the Keras model calls the backend (Tensorflow - as default - or Theano, etc.) and binds the optimizer, loss function, and other parameters.

More examples - Multi-Layer Perceptrons '**MLP**' in Kears on MNIST dataset - can be found either here - note 'reshape' of x data or in appendix [TO BE INSERTED].

Stochastic (mini-batch) gradient descent is very easy to implement in Keras. All we need to do is specify the size of the batches (batch_size) in the training process, as follows:

```
model.fit(X_train, y_train, epochs=1000,
batch_size=100, verbose=0)
```

Its learning rate needs to be small: it takes more time, but less likely to overshoot and missing the minimum. Sometimes, just decreasing such a rate is the solution to improve a poorly performing model.

Some terminology:

- one **epoch** = one forward pass and one backward pass of *all* the training examples;
- **mini-batch** size = the number of training examples in one forward/backward pass. The higher the mini-batch size, the more memory space you'll need.
- number of **iterations** = number of passes, each pass using [mini-batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your mini-batch size is 500, then it will take 2 iterations to complete 1 epoch.

Difficult bias-variance trade-off for neural network: increases **epochs** improves fitting (lower bias), but after some values it start to overfit the data. Some solutions for reducing overfitting are:

- **early stopping** algorithm in neural network: no. of epochs to be used are those where the testing error stops decreasing and starts increasing.
- **L-norm regularisations** (penalise larger weights): either (i) **L1** (i.e. by adding $\lambda \sum_i |w_i|$ to the error function - as in LASSO regression) which produces *sparse* (vector) weights and, hence, good for feature selection - or (ii) **L2** (i.e. by adding $\lambda \sum_i w_i^2$ - as in Ridge regression) which produces more *homogeneous* weights and whose gradient is smoother[4];
- **dropout** is a popular regularisation technique in neural networks: units (for each hidden or visible layer) are randomly ignored (dropped

[4] Think of a tangent to a diamond-shaped area in L1 vs. to a circle in L2 case.

out) during training (for each training sample, for each iteration), with probability p, so that a reduced network is left, avoiding too higher weights in some parts of the net. In Keras, we can easily add a dropout after adding a layer as follows:

```
    model.add(Dense(32, activation='sigmoid'))
    model.add(Dropout(0.2))
```

The parameter here that is set at 0.2, is for the probability that each node gets dropped of. Thus, as we make each training pass over the data, each node gets dropped off with a probability of 0.2.

Vanishing gradient issue: gradient may get quite small in neural networks (see fig ...).

INSERT FIGURE

One solution is to change the activation from (i) **sigmoid** function, $\sigma(x) = 1/(1 + e^{-x})$, to (ii) **hyperbolic tangent** function, $tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$, or to (iii) rectified linear unit (**ReLU**), $relu(x) = x$ if $x \geqslant 0$, 0 otherwise[5]. In Keras:

```
# examples of different activation fncts:
model.add(Activation('sigmoid')) # two classes
model.add(Activation('softmax')) # more classes
model.add(Activation('relu'))
model.add(Activation('tanh'))
```

In order to avoid being stuck in a local minimum with SGD, one can use:

- **random start**: simply start from different random places;
- **momentum**: exponential moving average of previous steps (fig. XXX);

INSERT FIGURE

There are many **optimisers** in Keras, most common being:

- Stochastic Gradient Descent: using (i) learning rate, (ii) momentum and (iii) Nesterov momentum (which lowers down the gradient when it is closer to the solution).

[5]Which is nothing more than $\max(0, x)$. This function is quite popular, more than the sigmoid - e.g ReLU in the intermediate layers and sigmoid in the output to get a probability.

- **Adam** (Adaptive Moment Estimation) uses a more complicated exponential decay that consists of not just considering the average (first moment), but also the variance (second moment) of the previous steps.
- **RMSProp** (Root Mean Squared Error prop) decreases the learning rate by dividing it by an exponentially decaying average of squared gradients.

[There are some other error functions that we did not touch].

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (ConvNets or CNNs) are a category of Neural Networks that have proven very effective in areas such as image recognition (identifying faces, handwritten digits, people/signs for self-driving cars, etc) and classification.

Contrary to MLPs where data are vectors (e.g. square pics of digits are flattened in vectors of black, grey and whites), CNNs really shines with real data, e.g. when images are not standardised as in the MNIST dataset.

Similarly MLPs and CNNs are composed of stacked layers (input, hidden, outputs) and use same optimisers for loss function, byt **types of hidden layers** for the latter are different (convolution, pooling, fully connected layers).

Hyperparameters search with Keras

Hyperparameter optimization is a big part of deep learning: neural networks are notoriously difficult to configure and there are a lot of parameters that need to be set (batch size and epoch, learning rate, activation function, weight initialisation, drop out, neurons in the hidden layers).

We can use the grid search (**GridSearchCV** class) from the scikit-learn library to tune the hyperparameters of Keras deep learning models, by wrapping such models with the **KerasClassifier** or **KerasRegressor** class.

One must provide a dictionary of hyperparameters to evaluate in the **param_grid** argument (array of values to try). By setting the n_jobs ar-

gument to -1, process will use all cores on your machine. Cross validation is used to evaluate each individual model and the default of 3-fold cross validation is used (although this can be overridden by specifying the **cv** argument).

Example to tune batch size and epoch number (source is here, also for more examples):

```
# Use scikit-learn to grid search the batch size and epochs
import numpy
from sklearn.model_selection import GridSearchCV
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
# Function to create model, required for KerasClassifier
def create_model():
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
# Compile model
# optimizer = SGD(lr=learn_rate, momentum=momentum)
model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])
return model

# fix random seed for reproducibility
numpy.random.seed(7)

# load dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")

# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]

# create model
model = KerasClassifier(build_fn=create_model, verbose=0)

# define the grid search parameters
batch_size = [10, 20, 40, 60, 80, 100]
epochs = [10, 50, 100]
param_grid = dict(batch_size=batch_size, epochs=epochs)

# other hyperparams ... need to change model fnct above
# optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta',
'Adam', 'Adamax', 'Nadam']
# learn_rate = [0.001, 0.01, 0.1, 0.2, 0.3]
```

```
# momentum = [0.0, 0.2, 0.4, 0.6, 0.8, 0.9]

grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1)
grid_result = grid.fit(X, Y)

# summarize results
print("Best: %f using %s" % (grid_result.best_score_,
grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']

for mean, stdev, param in zip(means, stds, params):
print("%f (%f) with: %r" % (mean, stdev, param))
```

Local connectivity

MLPs has two main **limitations**: (i) only use **fully** connected layers (hence a lot of parameters) and (ii) only accept **vectors** as inputs (losing the actual position of the pixels in a matrix). CNNs will address these issues by also (i) using **sparsely** connected layers (*locally* connecting some pixels to some hidden nodes, not all to all) and (ii) accept **matrices** as inputs - see fig. XXX.

INSERT FIGURE

Convolutional layers

filter are convolved over the image...
filter weights are learned to minimise a loss function ...

INSERT FIGURE

Adding more filters (tens to hundreds to..)

Source for the 4 filters (or activation maps) code below is here for more details.

```
# define 4 filters
import numpy as np

filter_vals = np.array([[-1, -1, 1, 1], [-1, -1, 1, 1],
```

```
[-1, -1, 1, 1], [-1, -1, 1, 1]])

filter_1 = filter_vals
filter_2 = -filter_1
filter_3 = filter_1.T
filter_4 = -filter_3
filters = [filter_1, filter_2, filter_3, filter_4]


fromfrom  keras.models  import Sequential
from keras.layers.convolutional import Convolution2D
import matplotlib.cm as cm

# plot image
plt.imshow(small_img, cmap='gray')

# define a neural network with a single convolutional layer with one filter
model = Sequential()
model.add(Convolution2D(1, (4, 4), activation='relu',
input_shape=(small_img.shape[0], small_img.shape[1], 1)))
# input_shape=(height, width, depth)

# apply convolutional filter and return output
def apply_filter(img, index, filter_list, ax):
# set the weights of the filter in the convolutional layer to filter_list[i]
model.layers[0].set_weights([np.reshape(filter_list[i], (4,4,1,1)),
np.array([0])])
# plot the corresponding activation map
ax.imshow(np.squeeze(model.predict(np.reshape(img, (1, img.shape[0],
img.shape[1], 1)))), cmap='gray')
```

In this car example, the first two filters identify the vertical lines (first the left line of the car), the other two horizontal lines.

Other hyperparameters of CCN (besides the required number and size of filters) are **stride**, amount of pixels that filter moves arounds (e.g. 1 pixel), and **padding** if filters strides outside (alternatively, ignore those nodes).

```
# import library: Convolution2D is an alias of Conv2D
from keras.layers import Conv2D

Conv2D(filters, kernel_size, strides, padding,
activation='relu', input_shape)
```

whose main arguments are:

- filters (required)- The number of filters;

- kernel_size (required) - an integer (tuple of 2 integers if different) specifying both the height and width of the convolution window (e.g. a square);
- input_shape (required - if first layer after input) - Tuple specifying the height, width, and depth (in that order) of the input.
- strides (optional)- The stride of the convolution, default is 1.
- padding (optional) - One of 'valid' or 'same', default is 'valid'.
- activation (optional) - typically 'relu'. default is that no activation is applied; but strongly encouraged to add a ReLU.

The **number of parameters** in a convolutional layer depends on the values of (i) filters (the number of filters in the convolutional layer, K), (ii) kernel_size (the height and width of the convolutional filters, F) and (iii) input_shape (the depth of the previous layer, i.e. last value in the input_shape tuple, D_{in}).
Since there are $F * F * D_{in}$ weights per filter and the convolutional layer is composed of K filters with one bias term per filter, then (the total number of weights is $K*F*F*D_{in}$ and) the number of parameters in the convolutional layer is given by $K * F * F * D_{in} + K$.

The **shape** (or spatial dimensions) in a convolutional layer depends on the values of (i) kernel_size (F), (ii) input_shape (H_{in} and W_{in} - height and with of the previous layer - respectively, the first and second value in the tuple), (iii) padding and (iv) stride (S):

- If $padding =' same'$, then the shape or spatial dimensions are
  - $height = ceil(float(H_{in})/float(S))$ and
  - $width = ceil(float(W_{in})/float(S))$
- if $padding =' valid'$, then the spatial dimensions are
  - $height = ceil(float(H_{in} - F + 1)/float(S))$ and
  - $width = ceil(float(W_{in} - F + 1)/float(S))$

The output shape in Keras is a tuple of (batch size=None, height, width, depth), where depth is always equal to the number of filters K.

Pooling layers

A third type of layers (besides Dense and Convolution ones) are pooling layers, that often take convolution layers as inputs - their goal is to reduce the number of parameters (and hence over-fitting) of convolutional layers.

There are two types of pooling layers: (i) **Max Pooling** Layer, for each feature map takes the max over a window (e.g. a square) striding e.g. 1 pixel at the time (generally half widht/height), and (ii) **Global Average Pooling** Layer, taking the average over feature maps (getting a vector, even a smaller reduction than max pooling).

```
from keras.models import Sequential
from keras.layers import MaxPooling2D

model = Sequential()
# MaxPooling2D(pool_size, strides, padding)
model.add(MaxPooling2D(pool_size=2, strides=2, input_shape=(100, 100, 15)))

model.summary()
```


where:

- pool_size (required)- Number specifying the height and width of the pooling window.
- strides (optional) - The vertical and horizontal stride, default is pool_size.
- padding (optional) - either 'valid' or 'same', default set to 'valid'.

CNNs for Image Classification

Resizing images to same size (usually square, divisible by a power of 2 - e.g. 32x32)

From one input layer to a sequence to convulation layers (common settings: strides=1, which is default, padding='same' which is not the default, activaltion='relu'), max pooling layer follows one or two conv layers (common settings: both pool_size and stride =2, padding default is ok - which halves spatial dimensions of conv layers).

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

```
3
4  model = Sequential()
5
6  # 3 conv. layers, followed by max pooling ones
7  model.add(Conv2D(filters=16, kernel_size=2, padding='same',
8  activation='relu', input_shape=(32, 32, 3)))
9  model.add(MaxPooling2D(pool_size=2))
10 model.add(Conv2D(filters=32, kernel_size=2, padding='same',
11 activation='relu'))
12 model.add(MaxPooling2D(pool_size=2))
13 model.add(Conv2D(filters=64, kernel_size=2, padding='same',
14 activation='relu'))
15 model.add(MaxPooling2D(pool_size=2))
16
17 model.add(Flatten())
18 model.add(Dense(500, activation='relu'))
19
20 model.add(Dense(10, activation='softmax'))
```

Always add a ReLU activation function (most of times is the best in practice) to the Conv2D layers in your CNN. With the exception of the final layer in the network, Dense layers should also have a ReLU activation function.

When constructing a network for classification, the final layer in the network should be a Dense layer with a softmax activation function. The number of nodes in the final layer should equal the total number of classes in the dataset.

For notebooks comparing between MLP and CNN on cifar10 (10 classes of tiny images, 32x32) here: from 40 to 66% accuracy improvement. The Kaggle winner (a CNN) can be found here.

Image Augmentation (Keras)

We want our model to learn an **invariant representation** of the image/object, one that is not influenced by size (scale invariant), angle (rotation invariance), position of the object in the image (translation invariance[6], e.g. squeezed to the left). One solution (which seems a bit like cheating) is to add some images with random rotation/translation/... to our dataset (this

[6]Max Pooling layers add some translation invariance to the model - intuitevel, max function tend to stay the same around a neighbourhood area.

is called **data augmentation**), moreover that makes the model also less prone to over-fitting too - we say that we have *augmented* our dataset.

The process starts by creating an **ImageDataGenerator** and then **fit** it on the data:

```
from keras.preprocessing.image import ImageDataGenerator

# create and configure augmented image generator
datagen = ImageDataGenerator(featurewise_center=True,
featurewise_std_normalization=True,
rotation_range=90.,
# randomly shift images horizontally (10% of total width)
width_shift_range=0.1,
# randomly shift images vertically (10% of total height)
height_shift_range=0.1,
zoom_range=0.2))

# then fit to data
datagen.fit(x_train)
```

The data generator itself is an iterator, returning batches of image samples when requested. We can configure the batch size and prepare the data generator and get batches of images by calling the **flow** function. Finally - instead of calling the fit() function on our model - we must call the **fit_generator()** function and pass in the data generator and the desired length of an epoch as well as the total number of epochs on which to train:

```
# model architecture
model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same',
activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))
# etc. ... ...
model.add(Flatten())
model.add(Dense(10, activation='softmax'))

# compile model
model.compile(...)

# get batches
batch_size = 32
x_batch, y_batch = datagen.flow(x_train, y_train, batch_size=32)

# instead of fit, use fit_generator
_steps_per_epoch = x_train.shape[0] / batch_size

```

```
model.fit_generator(x_batch, y_batch,
steps_per_epoch=_steps_per_epoch,
epochs=50)
```

where x_train.shape[0] (or len(train)) corresponds to the number of *unique* samples in the training dataset x_train. By setting steps_per_epoch to this value, we ensure that the model sees x_train.shape[0] augmented images in each epoch.

More details can be found on this blog post.

Groundbreaking CNN architectures

- AlexNext of Toronto Uni, 2012: introduced ReLU activation function and dropout (5 (11x11) convolutional layers)
- VGGNet, 2014 (Visual Geometry Group of Oxford) with either 16 or 19 (3x3) convolution layers, broken by 2x2 pooling layers, finished with 3 fully connected layers. Pioneered small windows (3x3) vs AlexNet larger ones.
- ResNet, 2015 (Microsoft) similarly VGG as idea of successive small layers (one was with 152 layers!), but with a solution to the vanishing gradient problem:

To recap - according to the universal approximation theorem - given enough parameters a feedforward network with a *single* layer is sufficient to represent any function. However, the layer might be massive and network prone to overfitting - hence, since AlexNet, the common trend for network architecture is to go *deeper* and deeper. Indeed, while AlexNet had only 5 convolutional layers, the VGG network had 19 layers and ResNet 152 (and up to 1k now). However, deep networks are hard to train because of the notorious *vanishing gradient problem* - as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitely small. As a result, as the network goes deeper, its performance gets saturated or even degraded. The core idea of ResNet is introducing a so-called *identity shortcut connection* that skips one or more layers - more on here.

You can access popular models for image classification already trained on ImageNet directly from keras.applications.

Transfer Learning

Transfer learning involves taking a pre-trained neural network and adapting the neural network to a new, different data set.

Depending on both (i) the **size** of the new data set, and (ii) the **similarity** of the new data set to the original data set - there are 4 approaches for using transfer learning.

...

A large data set might have one million images, a small data two-thousand images; the dividing line is somewhat subjective - but overfitting is a concern when using transfer learning with a small data set.

Regarding similarity, images of dogs and images of wolves would be considered similar since sharing common characteristics, but flower images datasets would be different from a data set of dog images.

If the new data set is **small and similar** to the original training data: (i) slice off the end of the neural network, (ii) add a new fully connected layer that matches the number of classes in the new data set, (iii) randomize the weights of the new fully connected layer, freeze all the weights from the pre-trained network (iv) train the network to update the weights of the new fully connected layer. To avoid overfitting on the small data set, the weights of the original network will be held constant rather than re-training the weights. Since data sets are similar, images from each data set will have similar higher level features, therefore most or all of the pre-trained neural network layers should be kept.

If the new data set is **small and different** from the original training data: (i) slice off most of the pre-trained layers near the beginning of the network, (ii) add to the remaining pre-trained layers a new fully connected layer that matches the number of classes in the new data set, (iii) randomize the weights of the new fully connected layer, freeze all the weights from the pre-trained network, (iv) train the network to update the weights of the

new fully connected layer. Because the data set is small, overfitting is still a concern: the weights of the original neural network will be held constant, like in the previous case - but since original and new data set do not share higher level features, the new network will only use the layers containing lower level features.

If the new data set is **large and similar** to the original training data: (i) remove the last fully connected layer and replace with a layer matching the number of classes in the new data set (ii) randomly initialize the weights in the new fully connected layer, (iii) initialize the rest of the weights using the pre-trained weights, (iv) re-train the *entire* neural network. Overfitting is not as much of a concern, therefore, you can re-train all of the weights. Also since datasets share higher level features, the entire neural network is used as well.

If the new data set is **large and different** from the original training data: (i) remove the last fully connected layer and replace with a layer matching the number of classes in the new data set, (ii) retrain the network from scratch with randomly initialized weights, (iii) alternatively, you could just use the same strategy as the "large and similar" data case above. Even though the data set is different from the training data, initializing the weights from the pre-trained network might make training faster. So this case is exactly the same as the case with a large, similar data set.

Recurrent Neural Network (RNN)

...to add memory to time-series ...

LSTM

3.4.1 Deep Learning in Finance

INSERT ZACH COMMENTS on recent papers

- Dixon etc.
- Three more papers...

So far: *In previous experiments, we tried predicting price movements directly with RNNs and while the RNN outperformed other approaches on the in-sample period, it failed to meaningfully out perform a linear model*

Appendix A

Annex

A.1 Johansen cointegration: code

```
'''
function result = johansen(x,p,k)
% PURPOSE: perform Johansen cointegration tests
% -------------------------------------------------------
% USAGE: result = johansen(x,p,k)
% where:      x = input matrix of time-series in levels, (nobs x m)
%             p = order of time polynomial in the null-hypothesis
%                 p = -1, no deterministic part
%                 p =  0, for constant term
%                 p =  1, for constant plus time-trend
%                 p >  1, for higher order polynomial
%             k = number of lagged difference terms used when
%                 computing the estimator
% -------------------------------------------------------
% RETURNS: a results structure:
%          result.eig  = eigenvalues  (m x 1)
%          result.evec = eigenvectors (m x m), where first
%                        r columns are normalized coint vectors
%          result.lr1  = likelihood ratio trace statistic for r=0 to m-1
%                        (m x 1) vector
%          result.lr2  = maximum eigenvalue statistic for r=0 to m-1
%                        (m x 1) vector
%          result.cvt  = critical values for trace statistic
%                        (m x 3) vector [90% 95% 99%]
%          result.cvm  = critical values for max eigen value statistic
%                        (m x 3) vector [90% 95% 99%]
%          result.ind  = index of co-integrating variables ordered by
%                        size of the eigenvalues from large to small
```

```
% ------------------------------------------------------
% NOTE: c_sja(), c_sjt() provide critical values generated using
%       a method of MacKinnon (1994, 1996).
%       critical values are available for n<=12 and -1 <= p <= 1,
%       zeros are returned for other cases.
% ------------------------------------------------------
% SEE ALSO: prt_coint, a function that prints results
% ------------------------------------------------------
% References: Johansen (1988), 'Statistical Analysis of Co-integration
% vectors', Journal of Economic Dynamics and Control, 12, pp. 231-254.
% MacKinnon, Haug, Michelis (1996) 'Numerical distribution
% functions of likelihood ratio tests for cointegration',
% Queen's University Institute for Economic Research Discussion paper.
% (see also: MacKinnon's JBES 1994 article
% ------------------------------------------------------
% written by:
% James P. LeSage, Dept of Economics
% University of Toledo
% jlesage@spatial-econometrics.com

'''
import numpy as np
from numpy import zeros, ones, flipud, log
from numpy.linalg import inv, eig, cholesky as chol
from statsmodels.regression.linear_model import OLS

from coint_tables import c_sja, c_sjt

tdiff = np.diff

class Holder(object):
pass

def rows(x):
return x.shape[0]

def trimr(x, front, end):
if end > 0:
return x[front:-end]
else:
return x[front:]

import statsmodels.tsa.tsatools as tsat
mlag = tsat.lagmat

def mlag_(x, maxlag):
'''return all lags up to maxlag'''
return x[:-lag]

```

```
78 def lag(x, lag):
79 return x[:-lag]
80
81 def detrend(y, order):
82 if order == -1:
83 return y
84 return OLS(y, np.vander(np.linspace(-1,1,len(y)), order+1)).fit().resid
85
86 def resid(y, x):
87 r = y - np.dot(x, np.dot(np.linalg.pinv(x), y))
88 return r
89
90 def coint_johansen(x, p, k):
91 nobs, m = x.shape
92
93 #f is detrend transformed series, p is detrend data
94 if (p > -1): f = 0
95 else:   f = p
96
97 x   = detrend(x,p)
98 dx  = tdiff(x,1, axis=0)
99 #dx = trimr(dx,1,0)
100 z   = mlag(dx,k)#[k-1:]
101 z   = trimr(z,k,0)
102 z   = detrend(z,f)
103 dx = trimr(dx,k,0)
104 dx    = detrend(dx,f)
105 #r0t   = dx - z*(z\dx)
106 r0t   = resid(dx, z)  #diff on lagged diffs
107 #lx = trimr(lag(x,k),k,0)
108 lx = lag(x,k)
109 lx = trimr(lx, 1, 0)
110 dx    = detrend(lx,f)
111 #rkt   = dx - z*(z\dx)
112 rkt   = resid(dx, z)  #level on lagged diffs
113 skk   = np.dot(rkt.T, rkt) / rows(rkt)
114 sk0   = np.dot(rkt.T, r0t) / rows(rkt)
115 s00   = np.dot(r0t.T, r0t) / rows(r0t)
116 sig   = np.dot(sk0, np.dot(inv(s00), (sk0.T)))
117 tmp   = inv(skk)
118 #du, au = eig(np.dot(tmp, sig))
119 au, du = eig(np.dot(tmp, sig))  #au is eval, du is evec
120 #orig = np.dot(tmp, sig)
121
122 # Normalize the eigen vectors such that (du'skk*du) = I
123 temp   = inv(chol(np.dot(du.T, np.dot(skk, du))))
124 dt     = np.dot(du, temp)
125
126
```

```
#JP: the next part can be done much  easier
#       NOTE: At this point, the eigenvectors are aligned by column. To
#       physically move the column elements using the MATLAB sort,
#       take the transpose to put the eigenvectors across the row

#dt = transpose(dt)

# sort eigenvalues and vectors

#au, auind = np.sort(diag(au))
auind = np.argsort(au)
#a = flipud(au)
aind = flipud(auind)
a = au[aind]
#d = dt[aind,:]
d = dt[:,aind]

#%NOTE: The eigenvectors have been sorted by row based on auind and moved to array "d
#%      Put the eigenvectors back in column format after the sort by taking the
#%      transpose of "d". Since the eigenvectors have been physically moved, there is
#%      no need for aind at all. To preserve existing programming, aind is reset back
#%      1, 2, 3, ....

#d  =  transpose(d)
#test = np.dot(transpose(d), np.dot(skk, d))

#%EXPLANATION:  The MATLAB sort function sorts from low to high. The flip realigns
#%auind to go from the largest to the smallest eigenvalue (now aind). The original pr
#%physically moved the rows of dt (to d) based on the alignment in aind and then used
#%aind as a column index to address the eigenvectors from high to low. This is a doub
#%sort. If you wanted to extract the eigenvector corresponding to the largest eigenva
#%using aind as a reference, you would get the correct eigenvector, but with sorted
#%coefficients and, therefore, any follow-on calculation would seem to be in error.
#%If alternative programming methods are used to evaluate the eigenvalues, e.g. Frame
#%followed by a root extraction on the characteristic equation, then the roots can be
#%quickly sorted. One by one, the corresponding eigenvectors can be generated. The re
#%array can be operated on using the Cholesky transformation, which enables a unit
#%diagonalization of skk. But nowhere along the way are the coefficients within the
#%eigenvector array ever changed. The final value of the "beta" array using either me
#%should be the same.


#% Compute the trace and max eigenvalue statistics */
lr1 = zeros(m)
lr2 = zeros(m)
cvm = zeros((m,3))
cvt = zeros((m,3))
iota = ones(m)
t, junk = rkt.shape
```

```
for i in range(0, m):
tmp = trimr(log(iota-a), i ,0)
lr1[i] = -t * np.sum(tmp, 0)  #columnsum ?
#tmp = np.log(1-a)
#lr1[i] = -t * np.sum(tmp[i:])
lr2[i] = -t * log(1-a[i])
cvm[i,:] = c_sja(m-i,p)
cvt[i,:] = c_sjt(m-i,p)
aind[i]  = i
#end

result = Holder()
#% set up results structure
#estimation results, residuals
result.rkt = rkt
result.r0t = r0t
result.eig = a
result.evec = d  #transposed compared to matlab ?
result.lr1 = lr1   # trace statistic
result.lr2 = lr2   # eigen statistic
result.cvt = cvt   # critical vals lr1 %90,%95,%99
result.cvm = cvm   # critical vals lr2 %90,%95,%99
result.ind = aind
result.meth = 'johansen'

return result
```

A.2 Second appendix