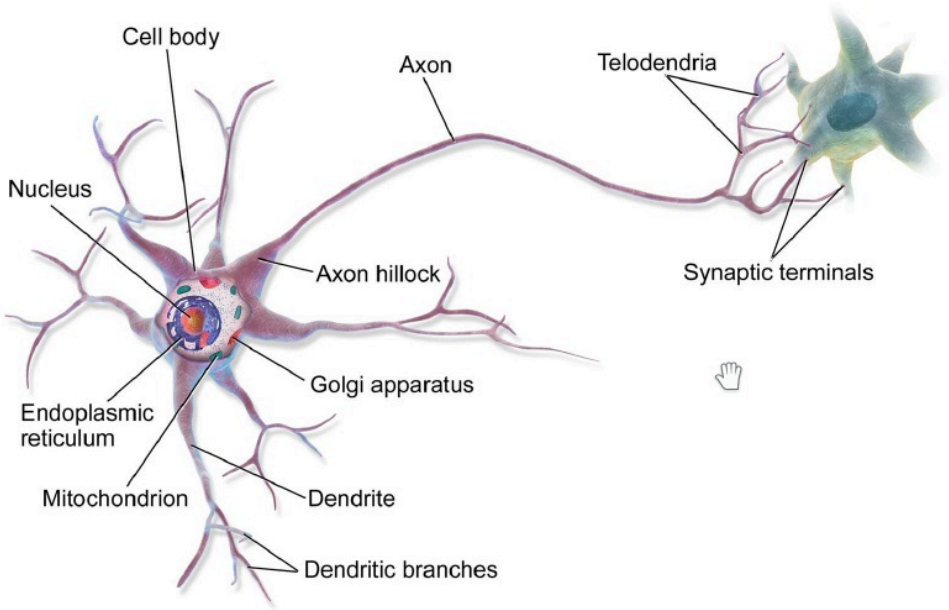
Functional Programming

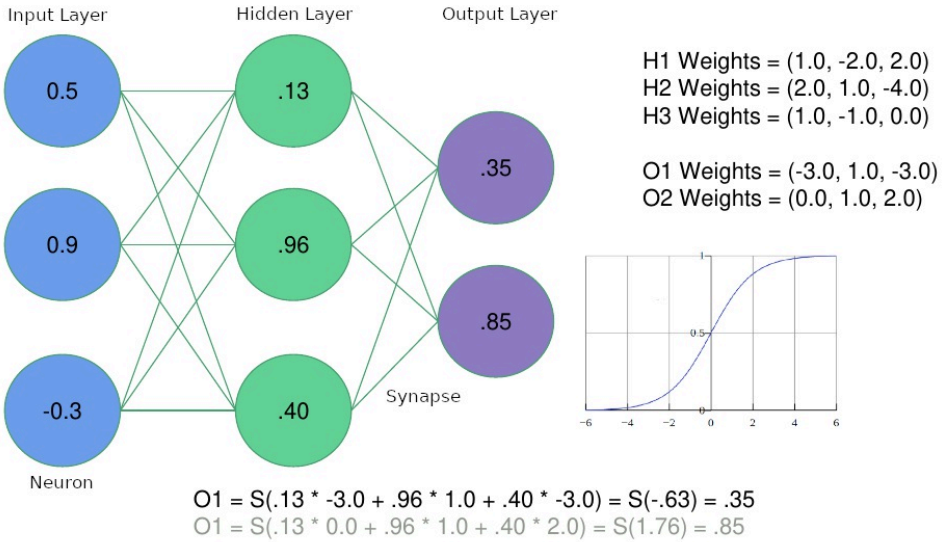
6. Differentiable Programming and Artificial Neural Networks

Frank C Langbein
frank@langbein.org

Version 1.4.0

Biological Inspiration






```
f_hidden :: R^n -> R^m,
f_hidden(x_1, ..., x_n) = S(W x)
  where    W is an (n, m) matrix of weights, one neuron per row
    and    S(y) = [s(y_1); ...; s(y_m)]   -- Activation functions
           where   s(y) = 1/(1+exp(-y))   -- Sigmoid
              or   s(y) = max(0,y)        -- ReLU
              or   s(y) = log(1+exp(y))   -- Softplus (SmoothReLU)
              ...
```

Training Neural Networks

- A neural network is a parameterised function approximator

```
f   :: A -> B
f_w :: A -> B for w in P
```

- Training
    - Take input/output samples `(a,f(a))` from `A x B`
    - Repeat until "learned":
        - Perform inference (compute `f_w(a)`) for training set
        - Calculate error / loss L between `f_w(a)` and `f(a)`
        - Update `w` according to learning rule based on error
- Typically `f_w` maps R^n to R^m (often `m << n`) and `w` are R^k weights (k large!)
    - Minimise mean squared error `1/n sum_k ||f(a_k) - f_w(a_k)||_2^2`
    - Or mean absolute error `1/n sum_k |f(a_k) - f_w(a_k)|`
    - Or cross entropy `sum_k f(a_k) log f_w(a_k)` (for probability distributions - see softmax)

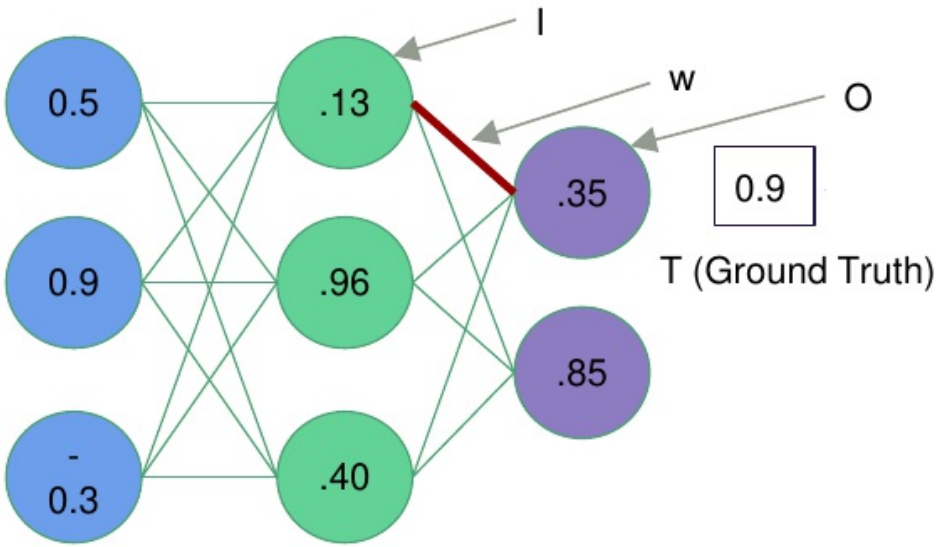
Backpropagation

- Which weights should be updated and by how much?
    - Use the derivative of the error with respect to weight to assign "blame"
- For one neuron its output 0 from inputs `I_k` with weights `w_k` is

```
O = s(N)   with  N = sum_k w_k I_k

s(y)  = 1      / (1+exp(-y))
      = exp(y) / (1+exp(y))
ds/dy = s(y) (1-s(y))
```

- So for a specific weight `w_k`

```
dL/dw_k = dL/dO  dO/dw_k = dL/dO  dO/dN  dN/dw_k

dL/dO   = (O-T)   -- for squared error
dO/dN   = s(N) (1-s(N)) = O (1-O)
dN/dw_k = I_k

dL/dw_k = (O - T) * O * (1-O) * I_k   = (.35-9) .35
(1-.35) .13
```



Gradient Descent

- Gradient descent minimizes the neural network's error
    - At each time step the error of the network is calculated on the training data
    - Weights are modified to reduce the error
    - Terminate when error is sufficiently small or max number of time steps has been exceeded
- Given the training samples, loss `L(w)` and gradient `D L(w)` (vector of partial derivatives w.r.t. weights) can be computed
    - Loss most rapidly decreases in negative gradient direction

```
w' = w - r DL(w)
   where r is the learning rate
```

- Typically stochastic gradient descent (e.g. SGD with momentum, AdaGrad, ADAM) are used

- Operate gradient descent on batches

Chain Rule

- Chain rule: `(f . g)' = (f' . g) * g'`
- A traditional neural network, organised in (fully connected) layers is a simple function composition

```
x_1   = f_1,w_1   x_0    = f_0   x_0
x_2   = f_2,w_2   x_1    = f_2   x_1
...
x_n-1 = f_n-1,w_n x_n-2  = f_n-1 x_n-2
x_n   = L(w)      x_n-1  = f_n   x_n-1
```

- The derivative w.r.t. the kth weight `w_k` for input `x_0`

```
D_k L x_0 = ((D_k f_n) x_n-1) * (D_k (f_n-1 x_n-2))
          = ((D_k f_n) x_n-1) * ((D_k f_n-1) x_n-2)  * (D_k (f_n-2 x_n-2))
          ....
          = ((D_k f_n) x_n-1) * ... * ((D_k f_1) x_0)
```

  - Assuming all derivatives exist (at least in some weak sense)

Modular System

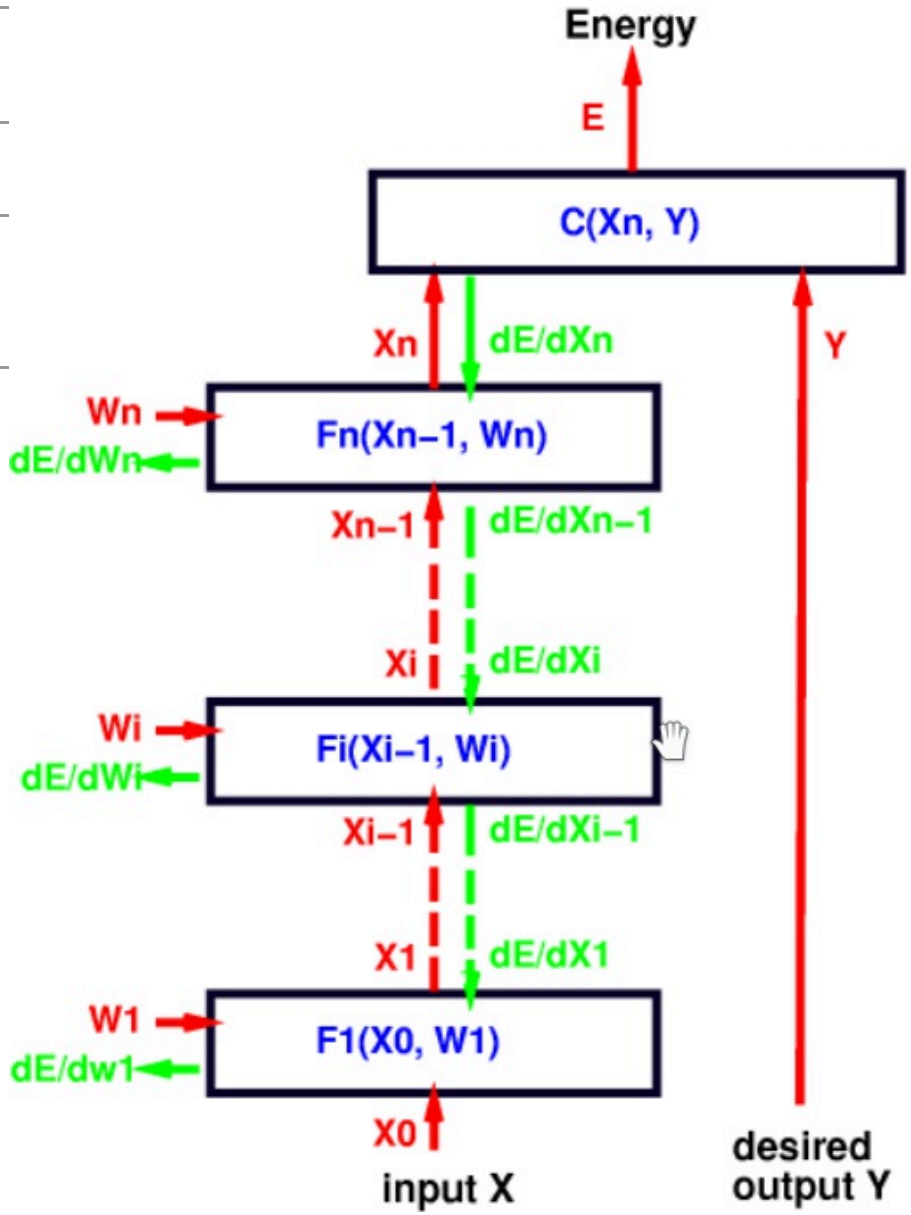
- Complex machines can be built by assembling modules
  - Simple example: layered feed-forward architecture (cascade)
- Forward propagation

```
let X_0 = X in
    X_i = F_l(X_i-1, W_i)  for i in [1,n]
    L(W) = E(Y,X,W) = C(X_n,Y)
```

- In Torch7

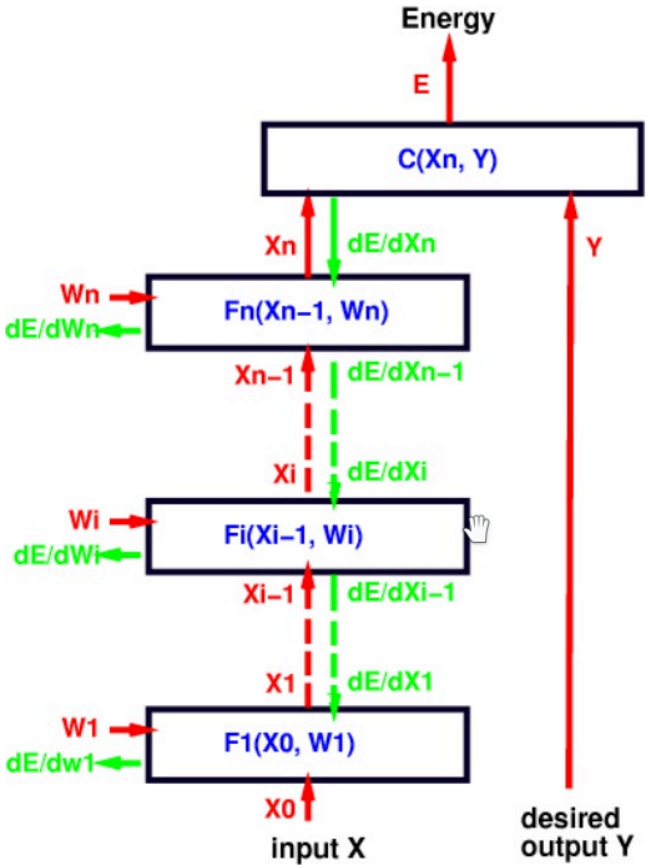
```
model = nn.Sequential()
model:add(m1)
model:add(m2)
out = model:forward(in)
```

- Each module is an object
    - contains trainable parameters
    - inputs are arguments
    - output is returned and stored internally



Gradient in Multi-Layer Systems I

- To train a multi-layer modular system, we compute the gradient of E with respect to all parameters `W_i`
- Module `i` computes `fprop`: `X_i = F_i(X_i-1,W_i)`
- Assume we know `dE/dX_i`:
    - for each `X_i[k]` we know how much E "wiggles" if we "wiggle" `X_i[k]`
- Chain rule:
  `dE/dW_i = dE/dX_i dF_i(X_i-1,W_i)/dW_i`
  `(1,N_w) = (1,N_x) (N_x,N_w)`
- `dF_i/dW_i` is the Jacobian matrix of `F_i` w.r.t. `W_i`:
  `JF_i = [ dF_i(X_i-1,W_i) / dW_i ]_kl =`
  `= d [ F_i(X_i-1,W_i) ]_k / d [ W_i ]_l`
    - JF_i[k,l] indicates how much the k-th output "wiggles" when we wiggle the `l`-th weight



Gradient in Multi-Layer Systems II

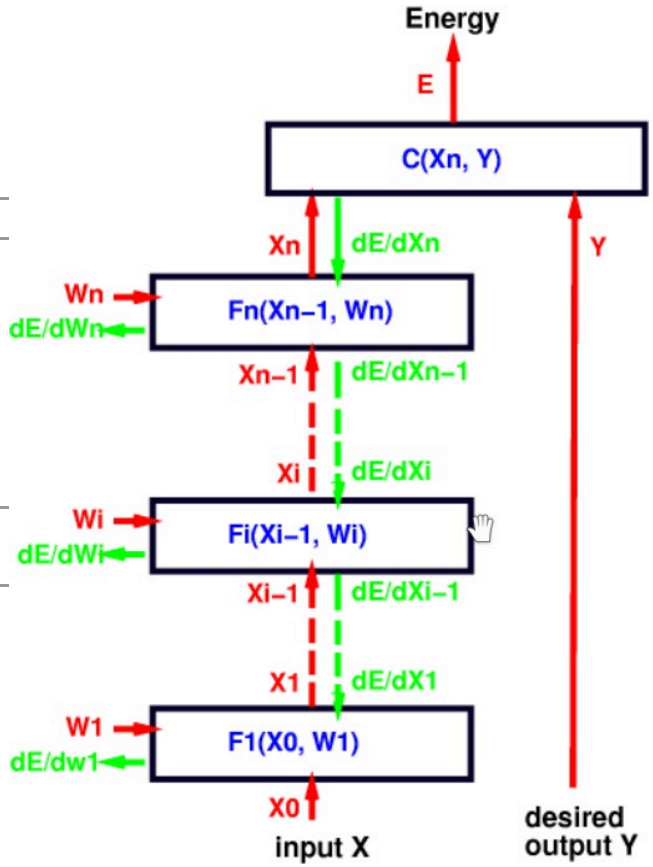
- Similarly, we compute `dE / dX_i-1`

- Assume we know dE / dX_i
    - How much does E "wiggle" is we "wiggle" X_i
- Chain rule:

```
   dE / dX_i-1 = dE / dX_i dF_i(X_i-1,W_i) / dX_i-1
```

    - dF_i/dX_i-1 is the Jacobian matrix of F_i w.r.t. X_i-1
    - F_i has two Jacobians as it has two arguments
    - Element [k,l] indicates how much the k-th output "wiggles" when we "wiggle" the l-th input
- Note, this is written with row vectors; transpose for column vectors

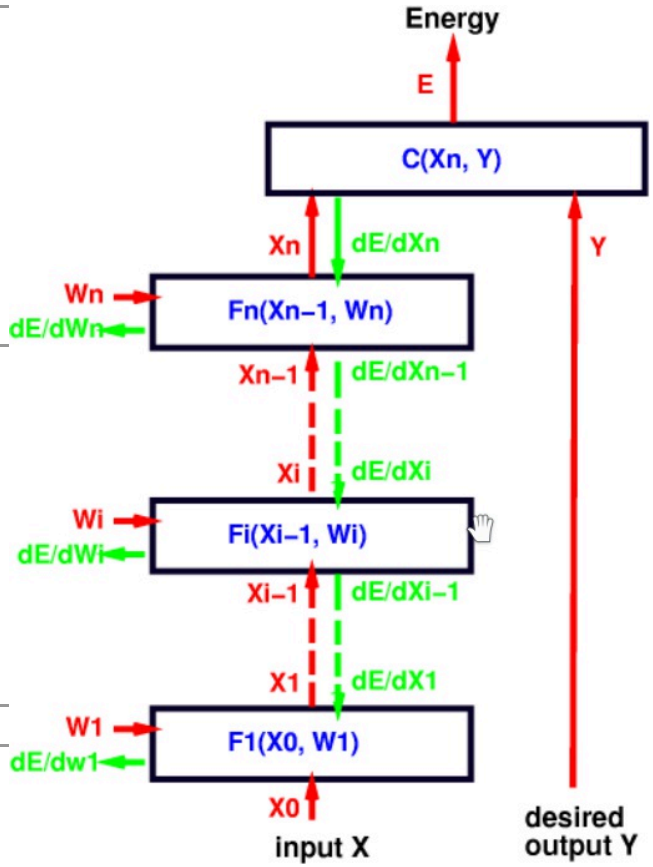
```
[ dE / dX_i ]' = [ dF_i / dX_i-1 ]' [ dE / dX_i ]'
[ dE / dW_i ]' = [ dF_i / dW_i   ]' [ dE / dX_i ]'
```



Backpropagation

- Backpropagation gives us all derivatives via a backward sweep via the recurrence equation for dE / dX_i:

```
DX_n   E = DX_n C(X_n,Y)

DX_n-1 E = DX_n   E  *  DX_n-1 F_n(X_n-1,W_n)
DW_n   E = DX_n   E  *  DW_n   F_n(X_n-1,W_n)

DX_n-2 E = DX_n-1 E  *  DX_n-2 F_n-1(X_n-2,W_n-1)
DW_n-1 E = DX_n-1 E  *  DW_n-1 F_n-1(X_n-2,W_n-1)

...until first module
```



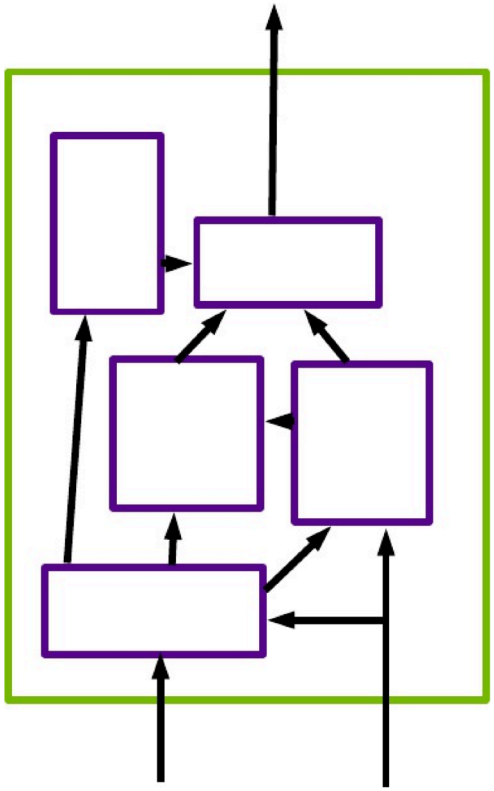
- This gives us all `DW_i E = dE / dW_i`
- Backpropagation through a module
  - contains trainable parameters
  - inputs are arguments and gradient w.r.t. output
  - gradient w.r.t. input is required
- Torch7 (with nn.Sequential)

```
ing = model:backward(in,outg)
```

Arbitrary Architectures

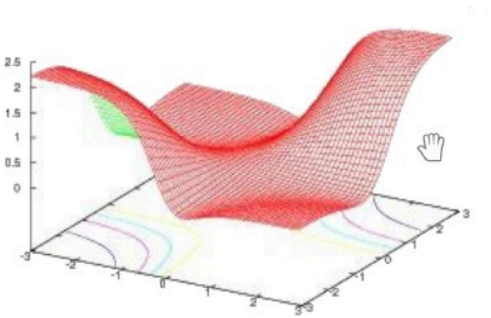
- Any connection between modules is permissible (given that input/output sizes match)
  - Networks with loops must be "unfolded in time"!
- Any module is permissible
  - As long as it is continuous and differentiable almost everywhere with respect to the parameters and the non-terminal inputs

- Automated differentiation
    - Calculate the derivatives of programs
    - Must be pure functions
    - Derivatives must exist (in some sense)
- Artificial Neural Networks
    - Match a function (defined by some notion of neurons) to a training set
    - Check that it generalises with a test set
- Combination allows to train arbitrary architectures



Backpropagation in Practice

- Deep supervised learning is non-convex!
    - Simplest 2-layer neural network:
      1-1-1 net: 0.5 to 0.5; -0.5 to -0.5
    - Loss: `0.5 - tanh(W_1 tanh(W_00.5))^2`
- Use ReLU non-linearities (tanh and sigmoid/logistic falling out of favour); cross-entropy loss for classification



- Use stochastic gradient descent on minibatches; shuffle training samples
- Schedule to decrease the learning rate; increasing batch sizes may be useful instead
- Normalise the input variables (zero mean, unit variance)
- Use a bit of L1/L2 regularisation on the weights (turn on after a couple of epochs)
- Use dropout for regularisation (and uncertainty estimation)
- More: LeCun et al "Efficient Backprop" (1998), Montavon, Orr, Muller, "Neural Networks, Tricks of the Trade" (2012)

Machine Learning

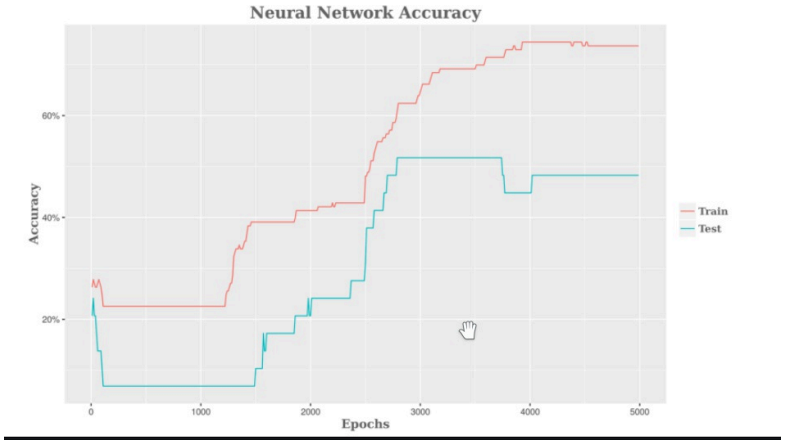
- Design a machine that can learn to perform a task
- Supervised learning
    - Labelled training data is provided
    - Goal: correctly label new data
- Reinforcement learning
    - Training data is unlabelled, provide feedback on performance/actions
    - Goal: perform better actions
- Unsupervised learning
    - Training data is unlabelled
    - Goal: categorise the observations

Measuring Success

- For classification
    - True positive: correctly identified as relevant
    - True negative: correctly identified as not relevant
    - False positive: incorrectly labelled as relevant
    - False negative: incorrectly labelled as not relevant
- `Precision = (true_positives) / (true_positives + false_positives)`
    - Percentage of positive labels that are correct
- `Recall = (true_positives) / (true_positives + false_negatives)`
    - Percentage of positive examples that are correctly labelled
- `Accuracy = (true_positives + true_negatives) / total_samples`
    - Percentage of correct labels
- Generalised to confusion matrix

Training and Test Data

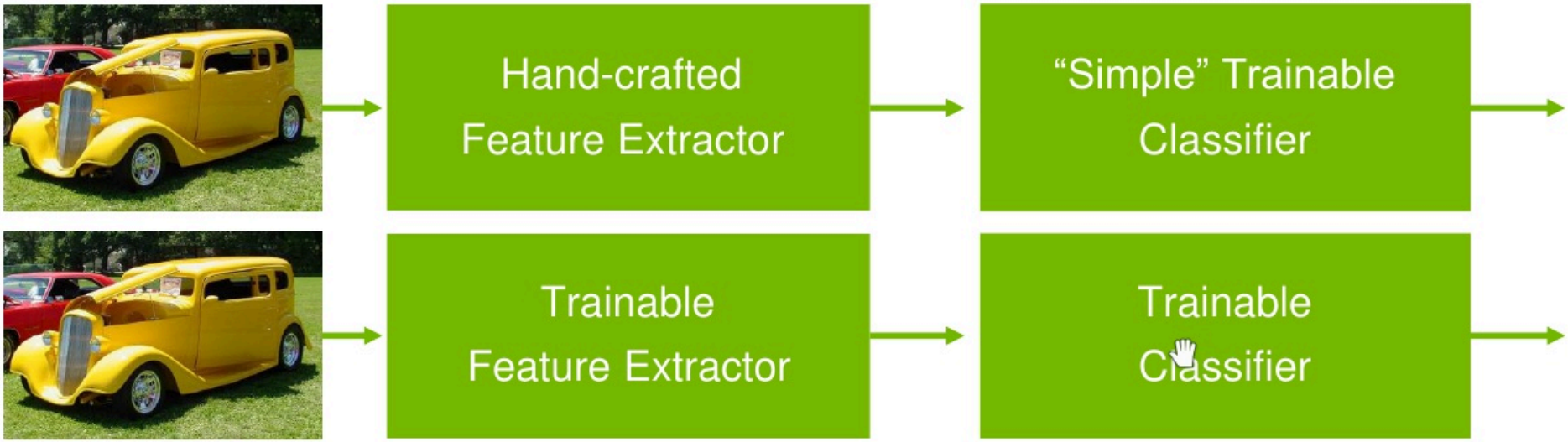
- Training data is used to learn a model
- Test data is used to assess the accuracy of the model
- Overfitting: model performs well on training data but poorly on test data
- Bias: expected difference between model's prediction and truth
- Variance: how much the model differs among training sets
- Model scenarios:
    - High bias: model makes inaccurate predictions on training data



- High variance: model does not generalise to new datasets
- Low bias: model makes accurate predictions on training data
- Low variance: model generalises to new datasets

Deep Learning

- Learning representations/features
- Traditional model of pattern recognition (since late 1950's)
    - Fixed/engineered features (or fixed kernel) + trainable classifier
- End-to-end learning / feature learning / deep learning
    - Trainable features (or kernel) + trainable classifier



Linear Machines - Regression with Mean Square

- Decision rule: `y = W' X`
- Loss function: `L(W,y^k,X^k) = 1/2 (y^k - W' X^k)^2`
- Gradient of loss: `DW L(W,y^k,X^k)' = -(y^k - W' X^k) X^k`
- Update rule: `W(t+1) = W(t) + r(t) (y^k - W(t)' X^k) X^k`
- Direct solution: `[sum_k X^k X^k'] W = sum_k y^k X^k`

Linear Machines - Perception

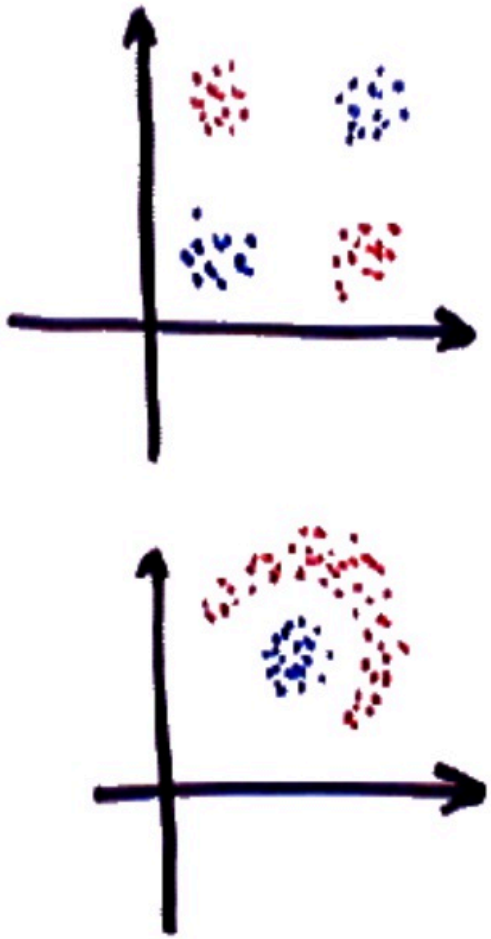
- Decision rule: `y = S(W' X)`
- Loss function: `L(W,y^k,X^k) = (S(W' X^k) - y^k) W' X^k`
- Gradient of loss: `DW L(W,y^k,X^k)' = -(y^k - S(W' X^k)) X^k`
- Update rule: `W(t+1) = W(t) + r(t) (y^k - S(W(t)' X^k)) X^k`
- Direct solution: find `W` such that `-y_k S(W' X^k) < 0` for all `k`

Linear Machines - Logistic Regression

- Decision rule: `y = S(W' X)` with `S(x) = tanh(x)` (sigmoid)
- Loss function: `L(W,y^k,X^k) = 2 log(1 - exp(-y^k W' X^k))`
- Gradient of loss: `DW L(W,y^k,X^k)' = -(y^k - S(W' X^k)) X^k`
- Update rule: `W(t+1) = W(t) + r(t) (y^k - F(W(t)' X^k)) X^k`

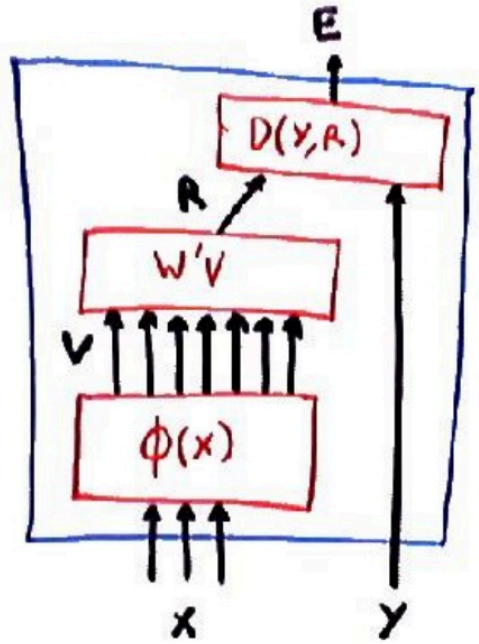
Limitations of Linear Machines

- Linearly separable dichotomies are partitions that are realisable by a linear classifier
    - The boundary between the classes is a hyperplane
- Cover's theorem, 1966:
    - The probability that P samples of dimension N are linearly separable goes to zero very quicly as P grows larger than N
    - There are 2^P possible dichotomies of P points, only about N are separable
    - If P is larger than N, the probability that a random dichotomy is linearly separable is very, very small
- Some simple dichotomies are not linearly separable



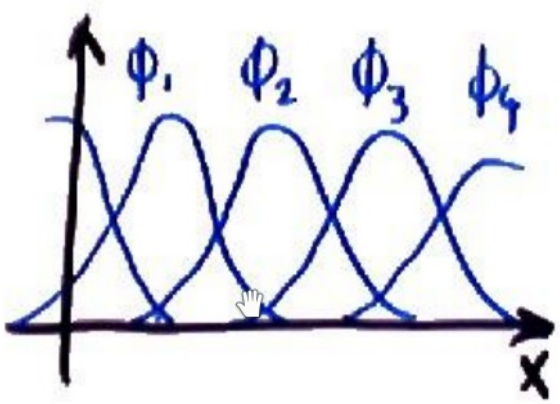
Idea 1: Make N Larger - Preprocessing

- Make N larger by augmenting the input variables with new "features"
    - Map X from its oriignal N-dimensional space into a higher dimensional space
    - Where things are more likely to be linearly separable
- Polynomial expansion: add quadratic terms (cross product of input vector)
    - From N to N(N+1)/2 dimensions
    - `phi(1,x_1,x_2) = (1,x_1,x_2,x_1^2,x_2^2,x_1 x_2)`
    - All conic sections are now linearly separable
    - Each conic boundary in the original space corresponds to a linear boundary in the transformed space
- Generalises poorly: scales as N^d - grows too quickly



Idea 2: Tile the Space

- Place a number of equally-spaced "bumps" that cover the entire input space
    - For classification: Gaussians
    - For regression: wavelets, sine/cosine, splines
- This does not work with more than a few dimensions
    - The number of bumps necessary to cover an N dimension space grows exponentially with N



Kernels - Sample-centred Basis Functions

- Place centre of a basis function around each training sample
    - Resources only spend on regions where there are samples
- Discriminant function:

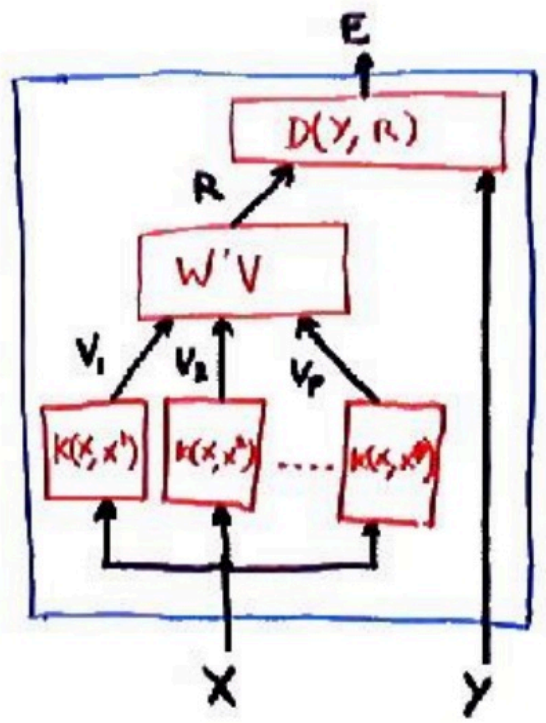
```
f(X,W) = sum_k W_k K(X,X^k)
```

- `K(X,X')` is often a radial basis function:

```
K(X,X') = exp(-b||X-X'||^2) -- Gaussian
K(X,X') = (X.X' + 1)^m      -- Polynomial
K(X,X') = tanh(XX')        -- sigmoid (single hidden-layer!)
```

- Common architecture, in particular for Support Vector Machines (SVM)
- Kernel centres may be reduced via clustering (e.g. k-means)

- Centres can be given by learnable parameters



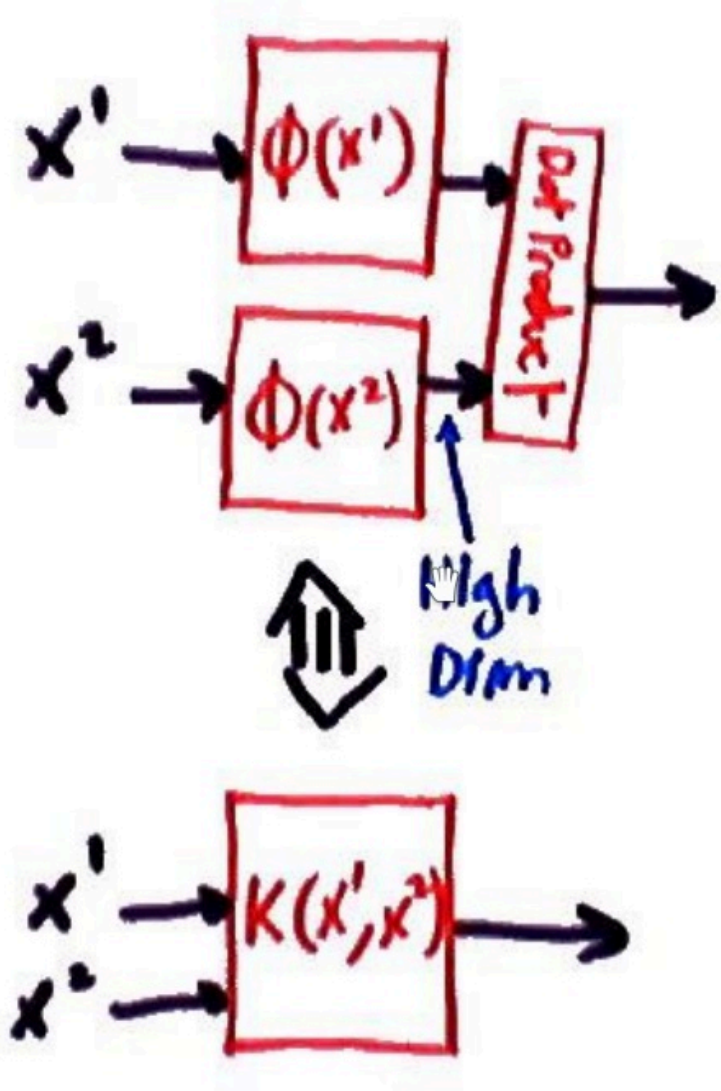
Kernel Trick

- If the kernel function `K(X,X')` fulfils the Mercer conditions, then there exists a mapping phi such that

```
phi(X) . phi(X') = K(X,X')
```

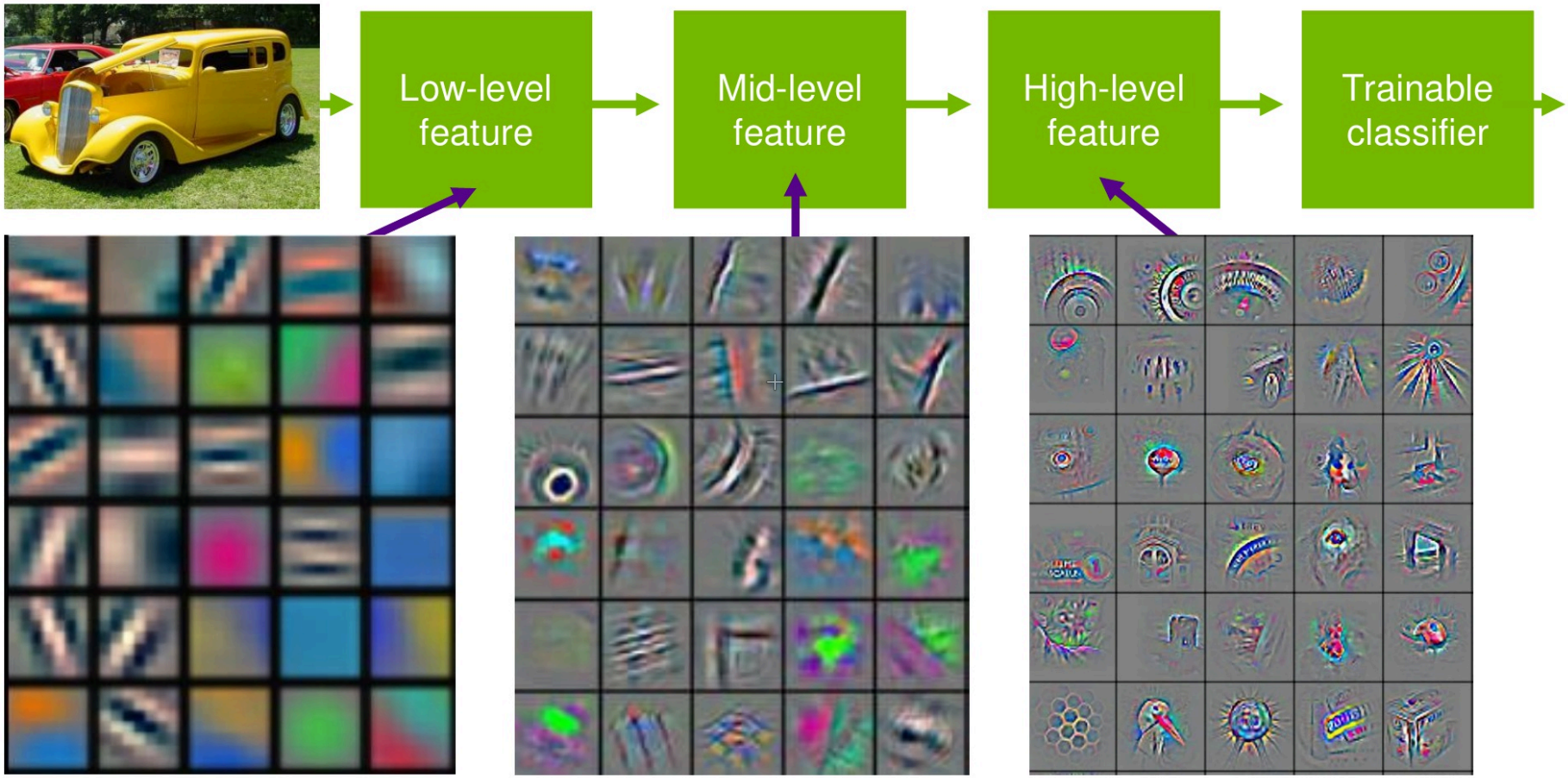
  - Mercer conditions: K is symmetric and positive definite
  - So we do not need to map X into the higher dimensional space explicitly

- Often combining modules in our in the network can be more efficient or numerically robust


x'
x²
φ(x')
φ(x²)
Dot Product
High Dim
x'
x²
K(x', x²)

Deep Learning = Learning Hierarchical Representations

- It's deep if it has more than one stage of non-linear feature transformation



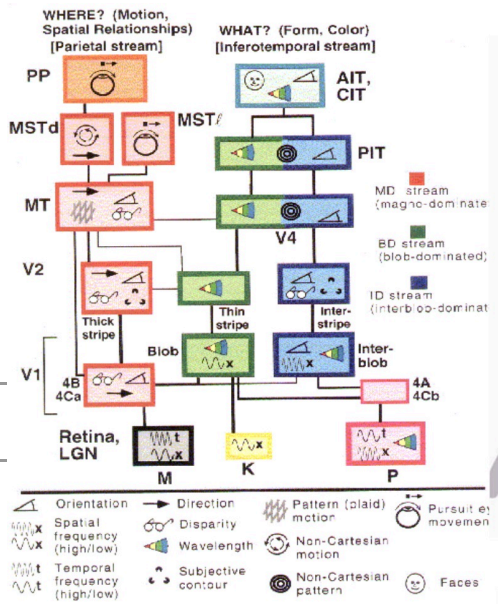
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

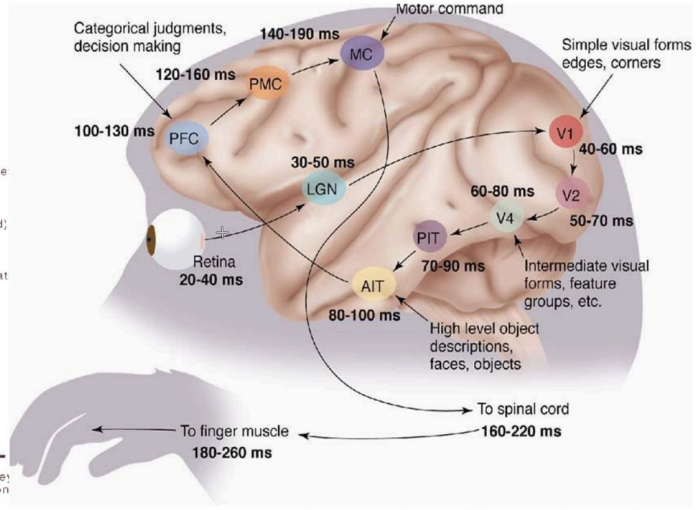
How Deep?

- The ventral (recognition) pathway in the visual cortex has multiple stages (Retina - LGN - V1 - V2 - V4 - PIT - AIT ....)
    - Lots of intermediate representations
    - (Also see HTMs)
- Theoretical result:
  We can approximate any function as close as we want with a shallow architecture

```
y = sum_k a_k K(X,X_k)                y = S(W_1 . S(W_0 . X))
```

    - Kernel machines and 2-layer neural networks are "universal"





[picture from Simon Thorpe]

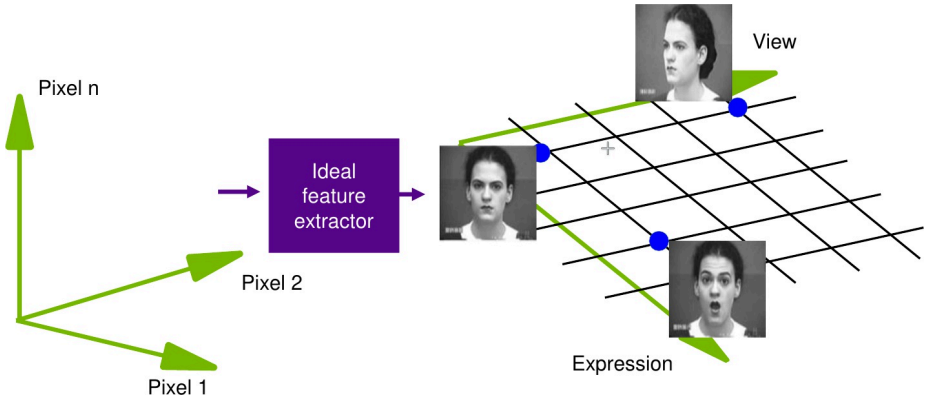
[Gallant & Van Essen]

- Deep machines are more efficient for representing certain classes of functions, particularly those involved in visual recognition
    - Represent more complex functions with less "hardware"
    - Efficient parametrisation of useful functions

The Manifold Hypothesis

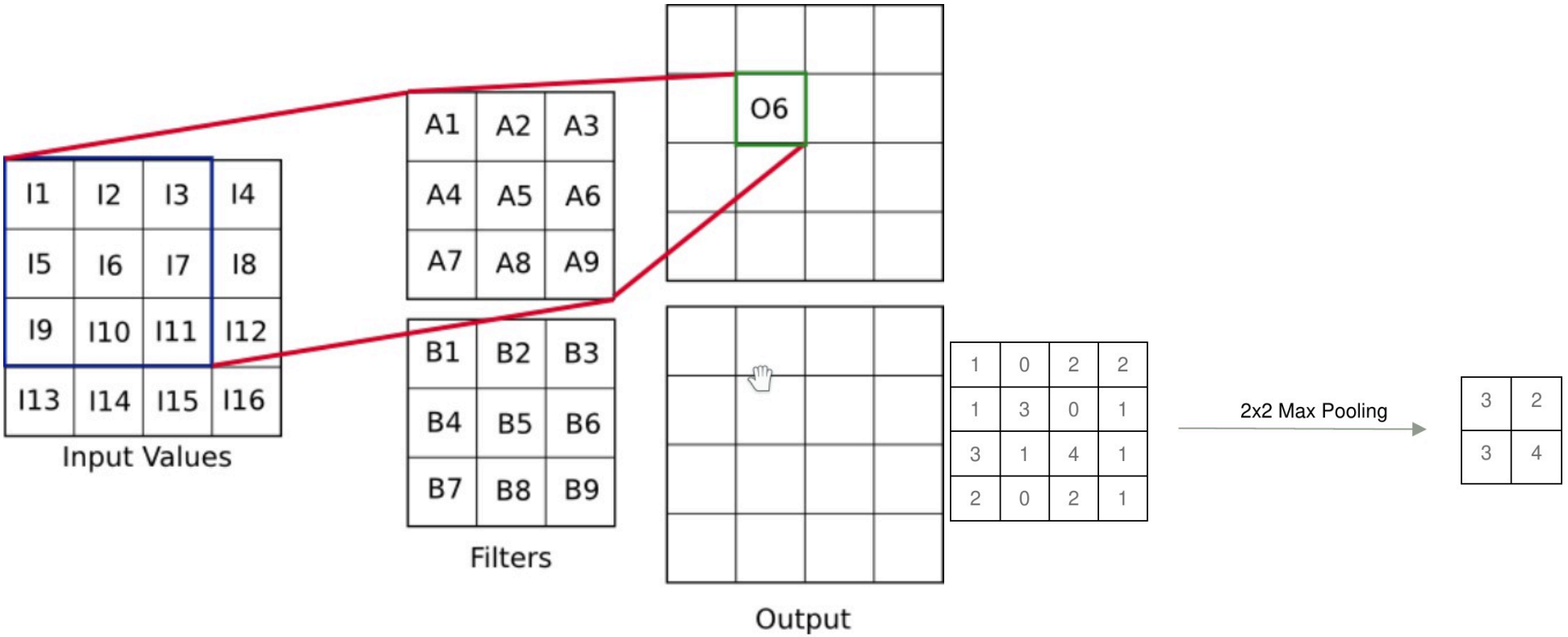
- Learning representations of data
    - Discovering and disentanlging the independent explanatory factors
- Manifold Hypothesis
    - Natural data lives in a low-dimensional (non-linear) manifolds

    - Because variables in natural data are mutually dependent
- Example: all face images of a person - 1000 x 1000 pixels = 3M bytes
    - But face has 3 Cartesian coordinates, 3 Euler angles and humans have less than about 50 muscles in the face
    - Manifold of face images for a person have < 56 dimensions
    - We just cannot find it (perfectly)



Convolutional and Pool Modules

- Instead of fully connected layers

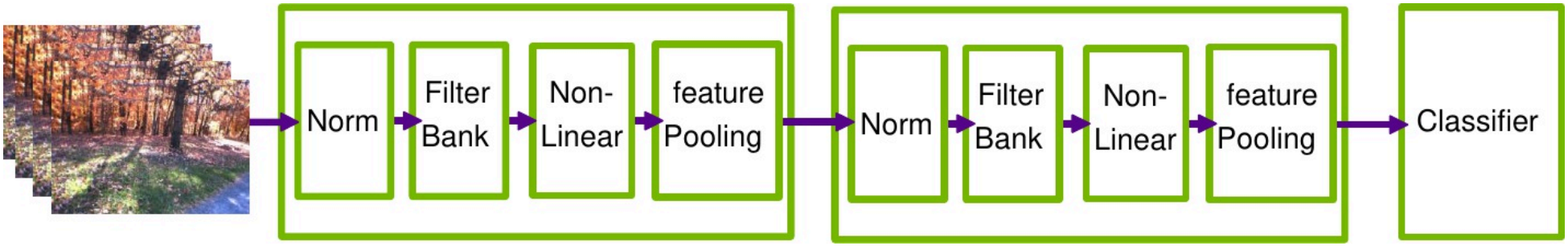


$$O_6 = A_1 \cdot I_1 + A_2 \cdot I_2 + A_3 \cdot I_3 + A_4 \cdot I_5 + A_5 \cdot I_6 + A_6 \cdot I_7 + A_7 \cdot I_9 + A_8 \cdot I_{10} + A_9 \cdot I_{11}$$

- Typically zero-filled at boundary

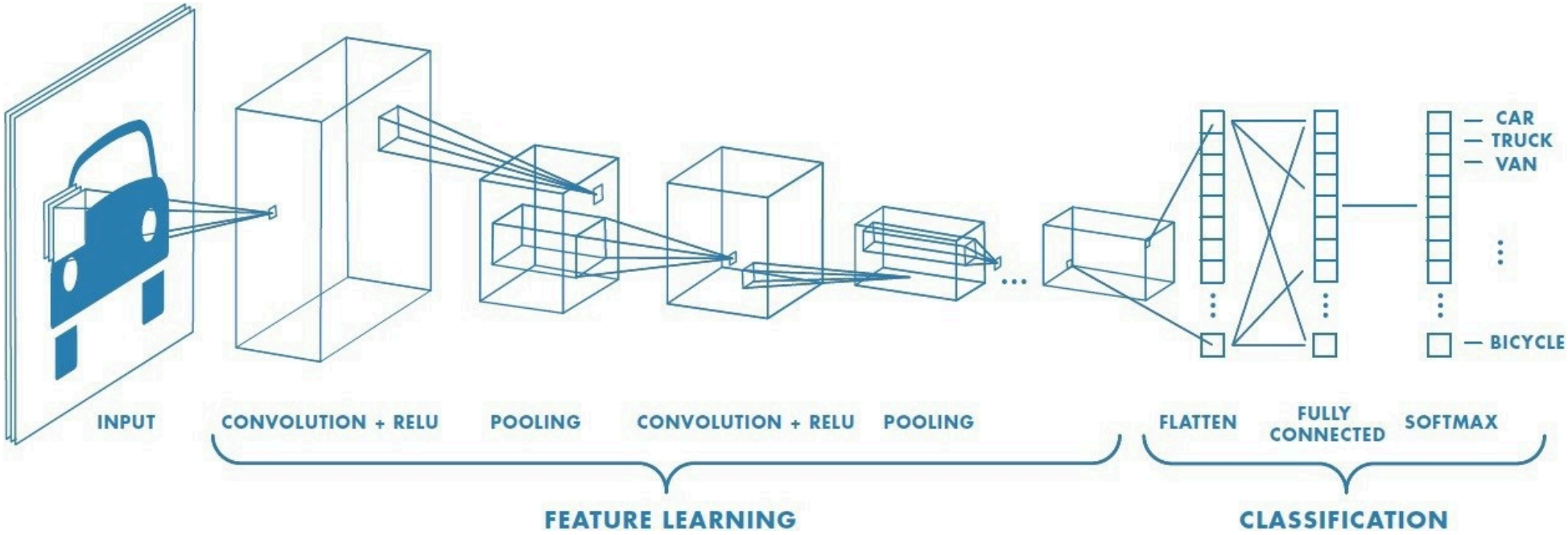
Convolutional Neural Networks

- Normalisation -> filter bank -> non-linearity -> pooling [repeat]



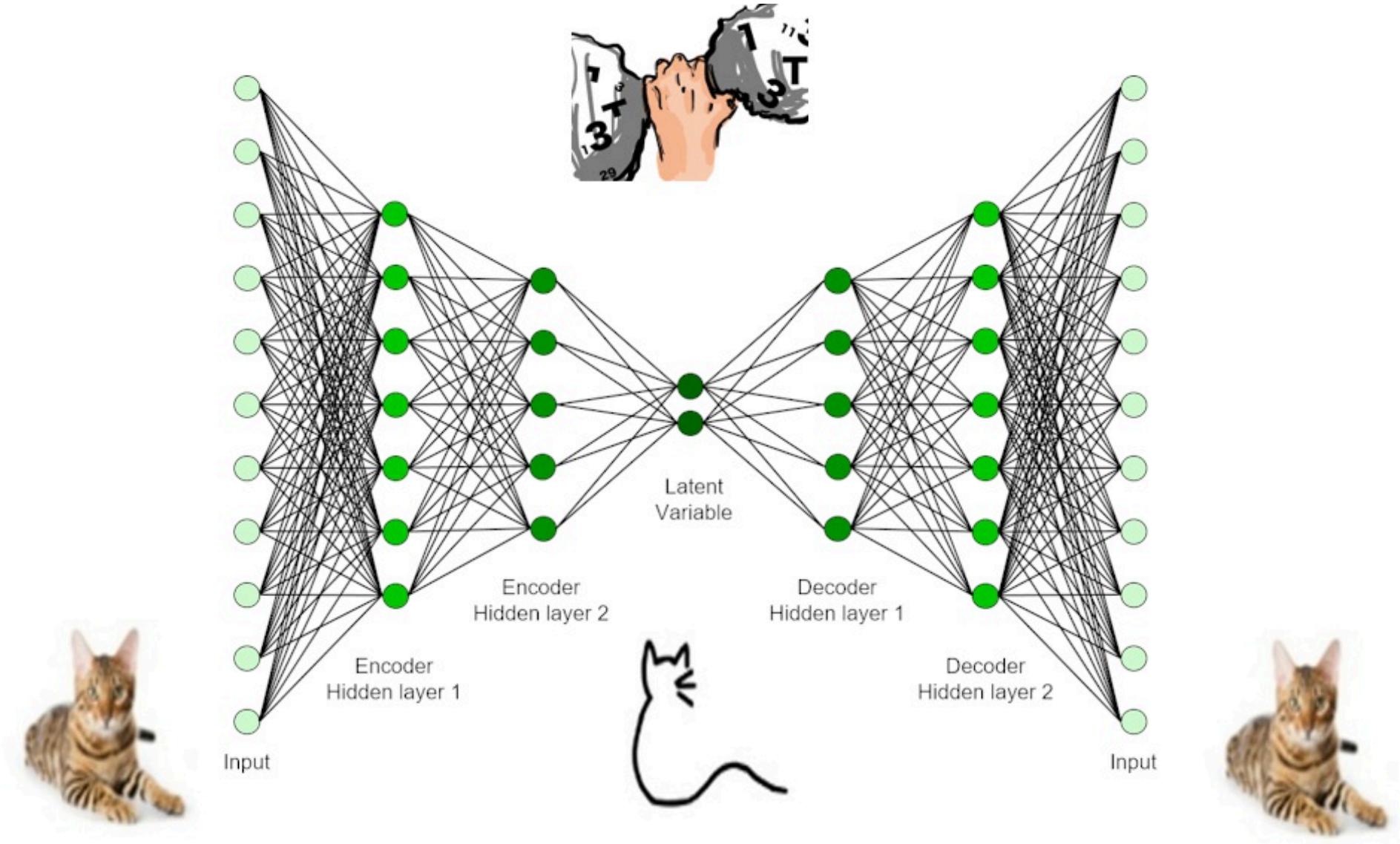
- Normalisation: variations on whitening
    - Subtractive: average removal, high-pass filtering
    - Divisive: local contrast normalisation, variance normalisation
- Filter bank: dimension expansion, projection on overcomplete basis
- Non-linearity: sparsification, saturation, lateral inhibition...
    - Rectification (ReLu), component-wise shrinkage, sigmoid, winner-takes-all
- Pooling: aggregation over space or feature type

CNN Architecture



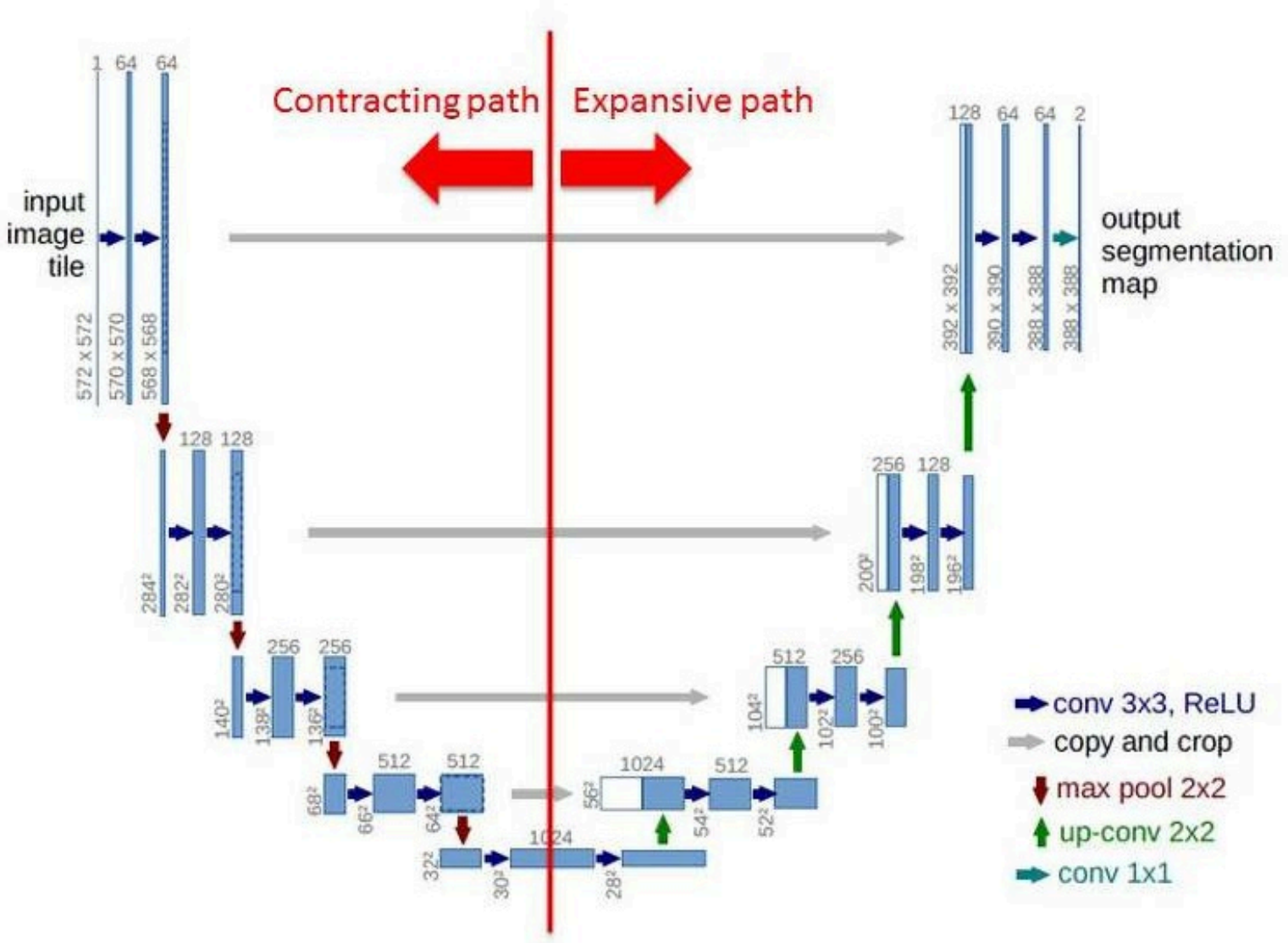
Sumit Saha, A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way, https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (2018)

Autoencoder Architecture



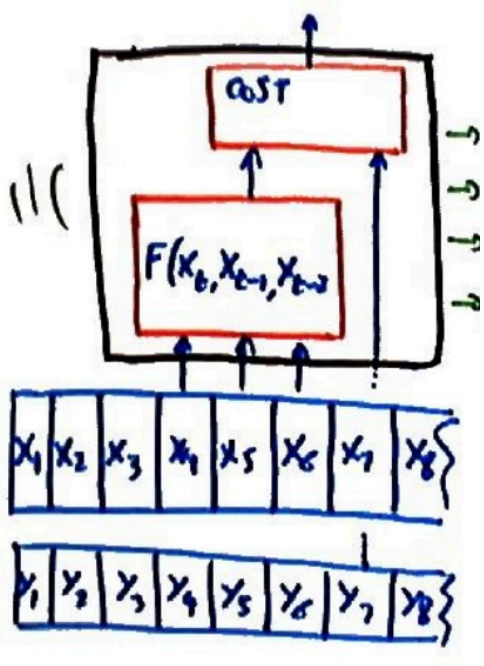
Chitta Ranjan, Extreme Rare Event Classification using Autoencoders in Keras, https://towardsdatascience.com/extreme-rare-event-classification-using-autoencoders-in-keras-a565b386f098 (2019)

U-Net Architecture

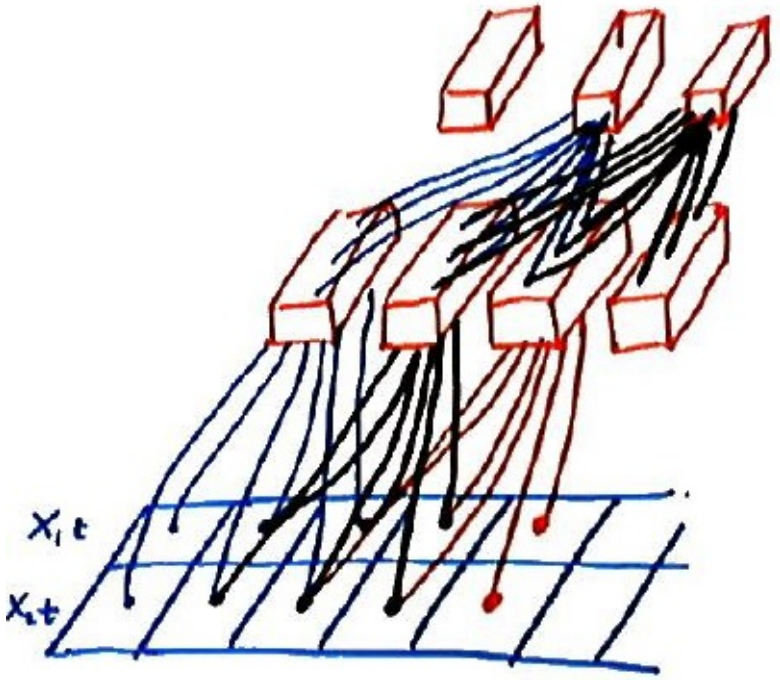


Sequence Processing

- Input is a sequence of vectors X_t
- The machine takes a time window as input
    - Predict the next sample in a time series (stock market, differential equations)
    - Predict next character or word in a text
    - Classify an intron/exon transition in a DNA sequence

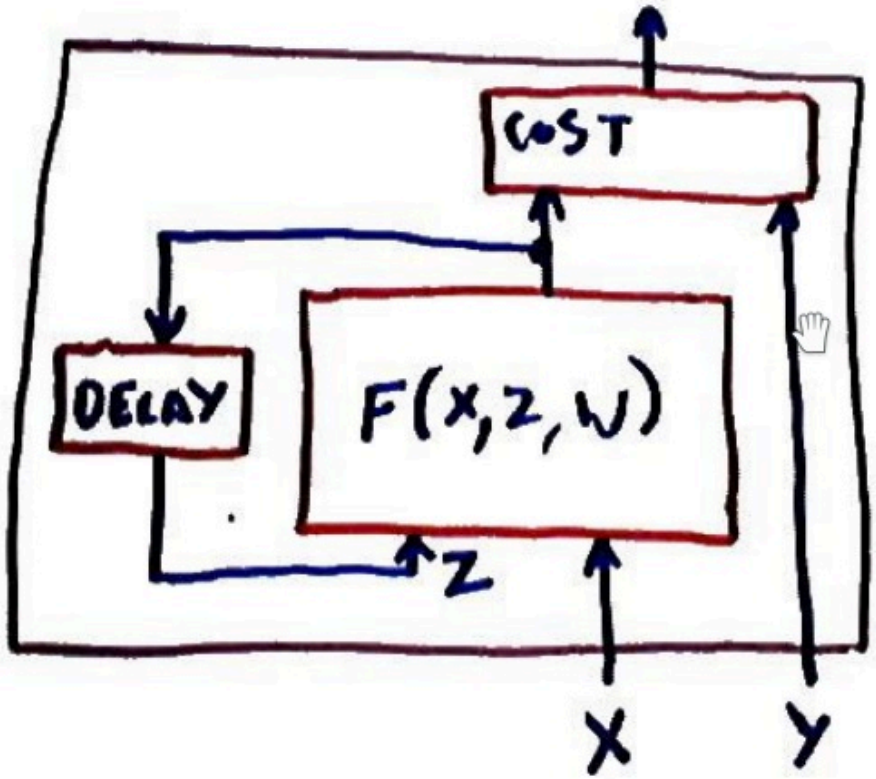


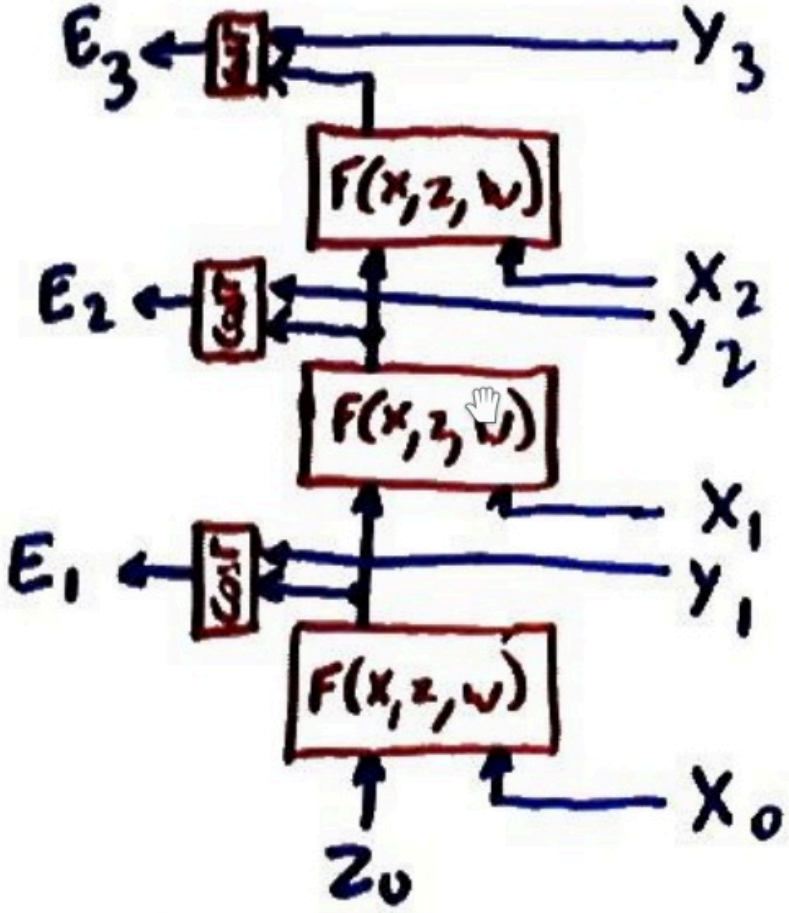
- Time delayed networks
    - One layer produces a sequences for the next layer
    - Minimal network influencing loss at time t is trained in isolation
    - Layers repeat over the time steps!
    - Sum over contributions of gradients from same module



Simple Recurrent Machines

- Output is fed back to some of the inputs
    - E.g. Hidden Markov Models (F is linear)
- Unfolding network creates a feed-forward net
    - Deep machine where all layers are identical with same weights
    - Can run backpropagation


COST
DELAY
F(X,Z,W)
Z
X
Y


E3
E2
E1
Y3
F(X,Z,W)
X2
Y2
F(X,Z,W)
X1
Y1
F(X,Z,W)
X0
Z0