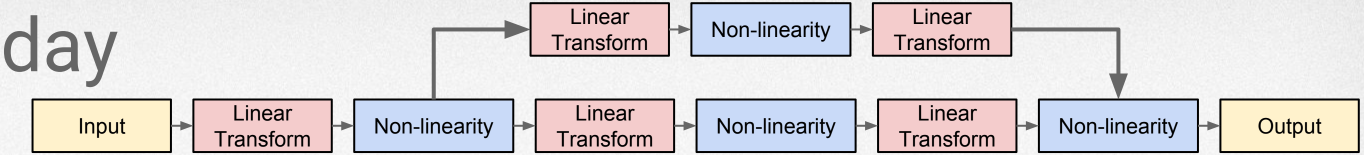
Neural Networks: Foundations

Simon Osindero

DeepMind

TL;DR for today

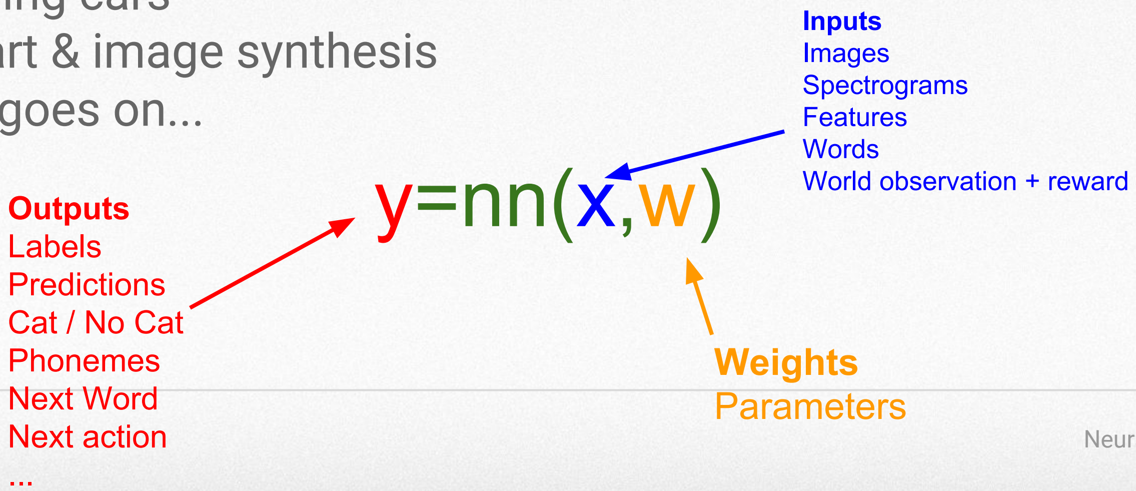


- NN’s: (more or less) just compositions of linear transforms + nonlinear functions.
    - Massive modelling power/flexibility by composing large graphs of these simple modules.
        - Routinely use 100M’s of parameters!
    - Learning → optimizing a loss function over data (or world experience) with respect to parameters.
        - Usually by chain rule derivatives and SGD.

What are neural nets good for?

Some recent successes

- Computer vision → objects, emotions, etc
- RNNs for speech recognition and (multi-lingual) machine translation
- Wavenets for text-to-speech
- DQN / A3C for reinforcement learning (e.g. Atari and more)
- AlphaGo
- Self-driving cars
- Neural art & image synthesis
- The list goes on...



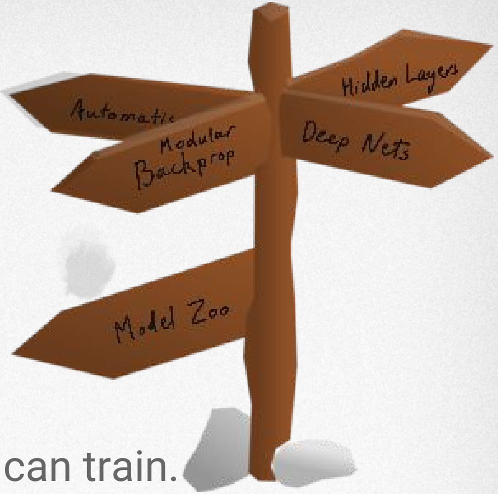
$$y=nn(x,w)$$

Inputs
Images
Spectrograms
Features
Words
World observation + reward

Outputs
Labels
Predictions
Cat / No Cat
Phonemes
Next Word
Next action
...

Weights
Parameters

Roadmap



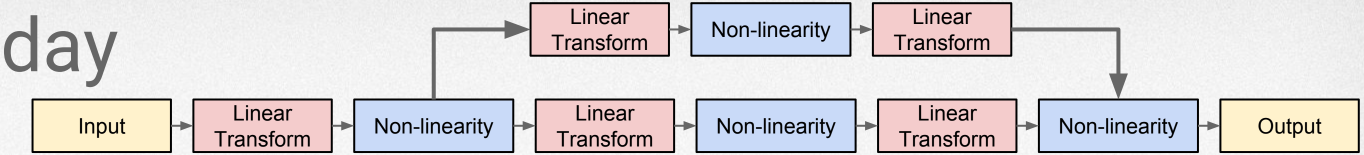
- Some things we won't cover
- Single layer networks
- Networks with one hidden layer
- Modern deep nets
  - Useful to think in terms of general compute graphs that we can train.
    - neural nets ↔ flexible, modular, non-linear functions
- Learning
  - Chain rule
  - Modular backprop & automatic differentiation
- Module zoo: layers and losses
- Discuss some tips for training models in practice
- [Time permitting] Backprop related research topic

Many different flavours of neural network

In this lecture series we aim to cover the subset most important for contemporary applications

- Large field with many branches
- Some interesting topics we wont cover
  - Boltzmann machines
  - Hopfield nets
  - Continuous time spiking neural net models
  - And many others…
- Take home:
  - We're focussing on the most important topics at the moment.
  - But encourage you to read widely and to keep an open mind.

TL;DR for today



Linear Transform

Non-linearity

Linear Transform

Input

Linear Transform

Non-linearity

Linear Transform

Non-linearity

Linear Transform

Non-linearity

Output

- NN's: (more or less) just compositions of linear transforms + nonlinear functions.
        - Massive modelling power/flexibility by composing large graphs of these simple modules.
        - Learning → optimizing a loss function over data (or world experience) with respect to parameters.
            - Usually by chain rule derivatives and SGD.

- **Caveats on terminology [this course & the field in general]**:
    - Sometimes use ***different* names** to refer to ***same* things**. E.g.
        - "neuron" aka "unit"
        - "non-linearity" aka "activation function"
    - Sometimes use ***same* name** to refer to ***different* things**. E.g.
        - Refer to compound (linear transform + non-linearity) as a "layer".
        - Refer to such atomic ops as a "layer" on their own (esp in modern software.)
    - Different semantic conventions may be when visualizing depicting models.
        - TF-style compute graphs vs more traditional sketches of layers

Basic single layer networks

Section Overview

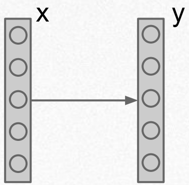
- “Real” neurons: NNs are **very** coarse approximation [we won’t dwell much here]
- Linear layer
- Sigmoid activation function layer
- Recap binary classification/logistic regression
- Multi-way decisions: softmax layer
- Rectified linear layers (relu)

Linear layers

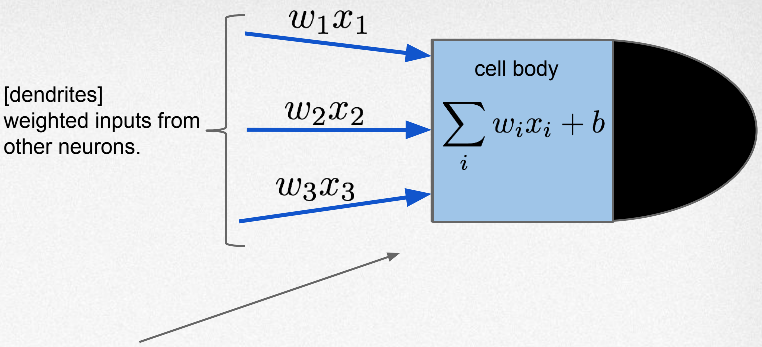
aka “fully connected” or “dense” layers

- Weighted sum of inputs (and bias term)
  - i.e. an affine transformation
  - $$y = \sum_i w_i x_i + b$$
- Describe “layer” of linear neurons with matrix-vector operations.

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

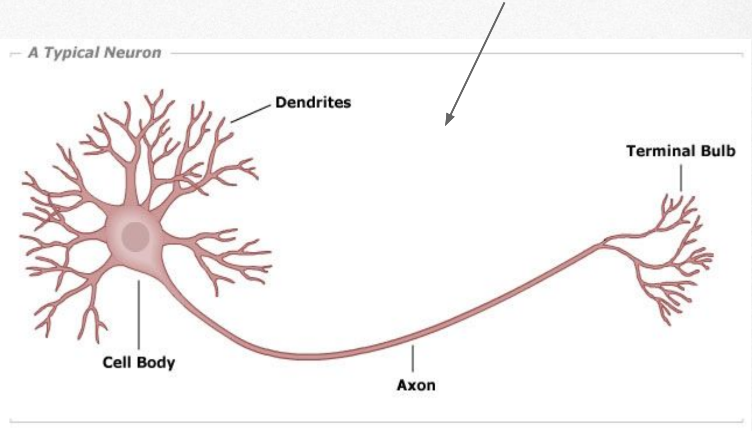


- Can be trained as a linear regressor (prev lecture).
- Typically follow this by an additional transformation using a non-linear “activation function”.



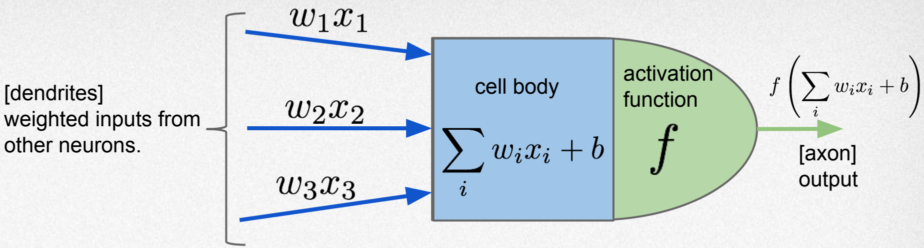
Simple linear “neuron”

“Real” neuron

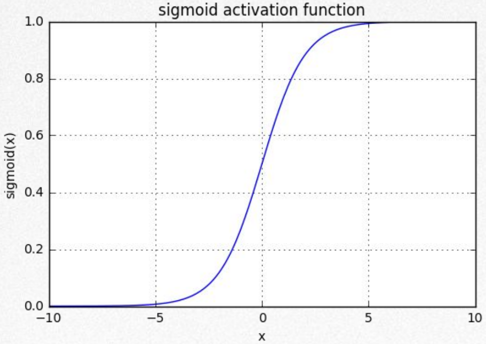


Sigmoid layer

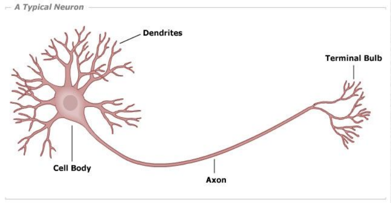
A type of “activation function”



- “Squashing” function.



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



- Typically applied after a linear layer
  - $$y = \sigma\left(\sum_i w_ix_i + b\right)$$

- Lots of freedom to choose what the activation function is.
- Best choice depends on specifics of problem we’re solving.
- We’ll see some common choices later.

Linear layer + sigmoid as a binary classifier

Logistic regression

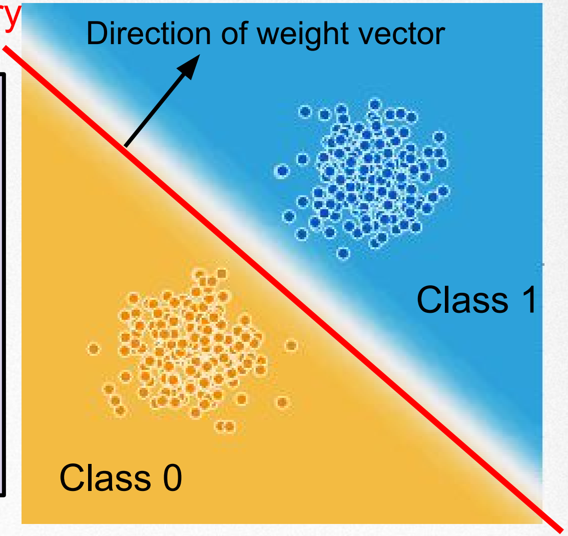
- Recall Thore’s class:

$$y = \sigma\left(\sum_i w_i x_i + b\right)$$

(Different notation to this class)

- Linear model: $z(\mathbf{w}, \mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$
- (Inverse) Link function: $\hat{p}(z) = \dfrac{1}{1 + \exp(-z)}$
- Cross entropy loss: $l(y, \hat{p}) = y \log \hat{p} + (1 - y) \log(1 - \hat{p})$
- The regression loss is a composition of these three functions, aggregated over training examples

Linear decision-boundary



http://playground.tensorflow.org/

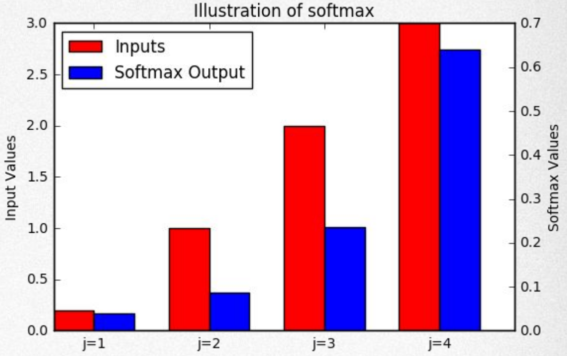
Softmax layers

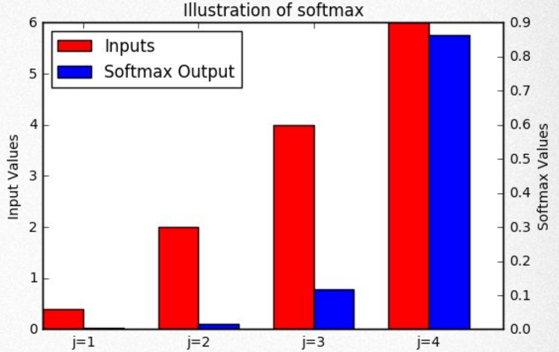
Useful in multi-class classification and also in soft/differentiably multi-way gating/routing

- Input vector **x.**
- Argmax function applied to **x:**
  - all but the largest element becomes 0;
  - the largest element becomes 1.
- Softmax performs a “soft” argmax instead.

$$\mathbf{y} = \mathrm{softmax}(\mathbf{x}), \text{ where: } y_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

- Elements of **y** are non-negative and sum to 1.
- Can interpret as a probability distribution over the K indices of **y**.





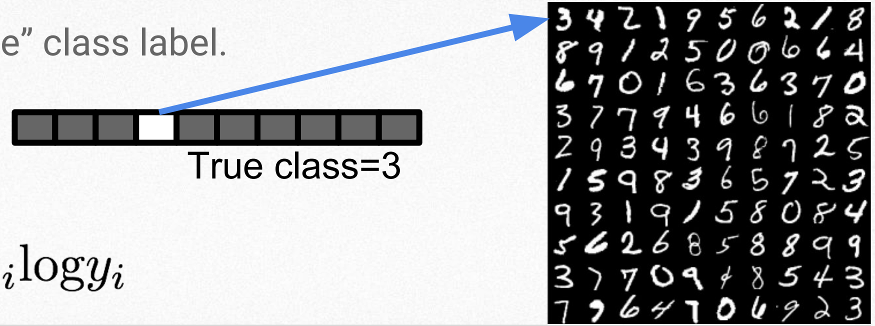
Softmax and cross-entropy / NLL loss

Useful in multi-class classification and also in soft/differentiably multi-way gating/routing

- Linear layer + softmax allows us to perform multinomial logistic regression / multi-class classification
  - Similar to what we saw with sigmoid and logistic regression/binary classification.

$$y_i = \frac{e^{\sum_j w_{ij} x_j + b_i}}{\sum_k e^{\sum_j w_{kj} x_j + b_k}}$$

- Again, we train by minimising the negative log likelihood (NLL) / cross-entropy of true labels under our predictive dist.
  - Use a “1-hot” vector, **t**, to encode the “true” class label.
    - **t_i=1 if true class label is i,**
    - **t_i=0 otherwise**



$$\mathrm{NLL}(\mathbf{t},\mathbf{y}) = \mathrm{Xent}(\mathbf{t},\mathbf{y}) = -\sum_{i}^{\mathrm{Classes:C}} t_i \log y_i$$

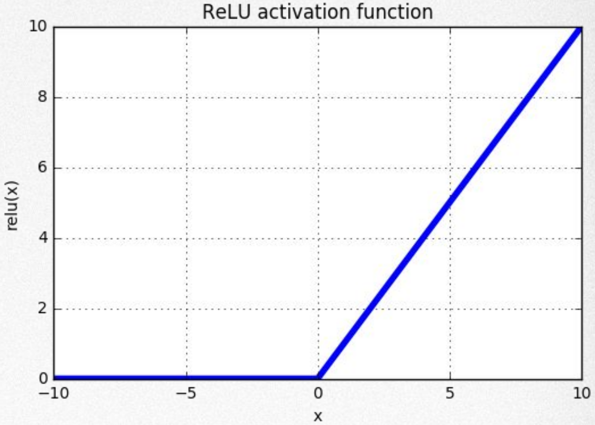
Rectified-linear layer

Gives us rectified linear units or ReLUs

- Simpler/cheaper than sigmoid.
- Prevalent in modern neural nets.

$$\mathbf{y} = \mathrm{relu}(\mathbf{x}), \text{ where: } y_i = \mathrm{maximum}(0, x_i)$$

- Favourable gradient properties can significantly help with learning
  - A little more about that later in this lecture.
  - Also Karen Simonyan & James Marten’s lectures.



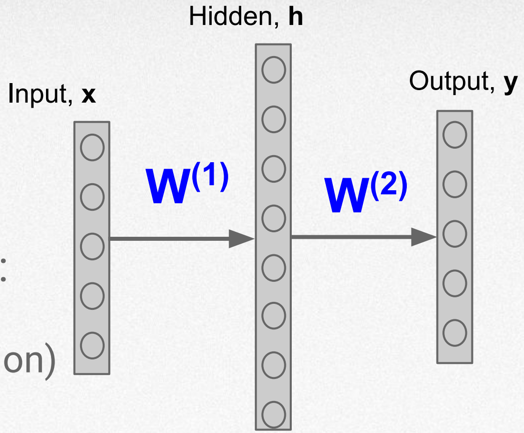
Networks with one hidden layer

Section Overview

- Hidden layers and binary classification when inputs not linearly separable.
- Hidden layer nets as “universal function approximators”.

- **Caveat again on terminology**:
    - Traditionally: hidden layer means (linear + non-linearity).
        - Will use that for this section.
    - More modern terminology often refers to linear/non-linear parts as separate layers (or modules)
        - Particularly common when we switch to the context of more general compute graphs.

Hidden layers



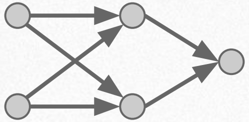
- Single hidden layer:
    - In this case, by “layer” we mean the compound:
        - linear module
        - followed by non-linear module (activation function)
        - (we’ll call this a linear+sigmoid layer)

- Outputs of one layer → inputs to the next.
- Allow us to transform input into intermediate representations.
    - Our final problem may be simpler in this transformed representation.
        - Recall the non-linear basis functions from Thore’s previous lecture.

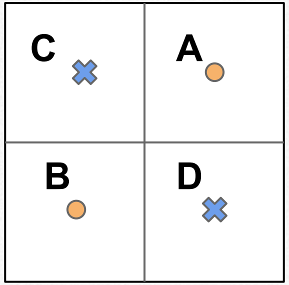
Logistic regression when inputs not linearly separable

e.g. simple XOR

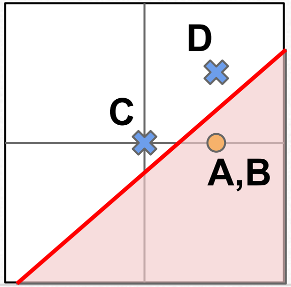
Point	Original Space	Hidden Space	
A	(1,1)	(1,0)	Class 0
B	(-1,-1)	(1,0)	
C	(-1,1)	(0,0)	Class 1
D	(1,-1)	(1,1)	

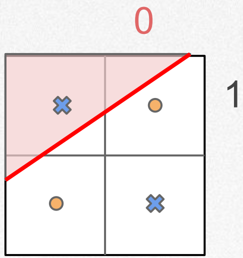


Input Layer

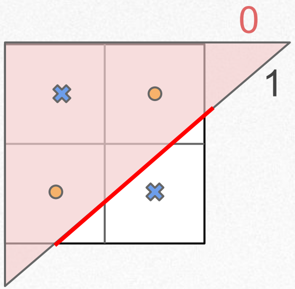


Hidden Layer





1st hidden unit

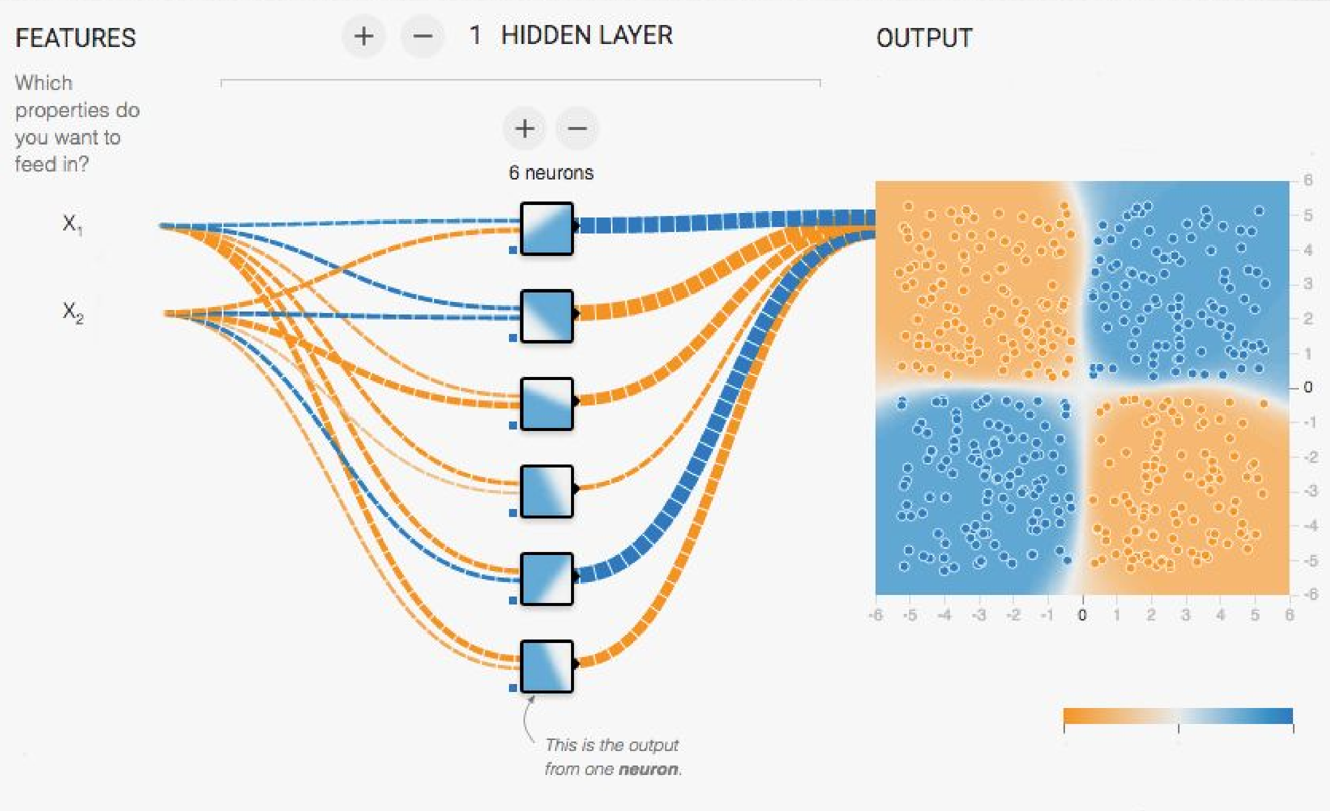


2nd hidden unit

Recommended check-out: Tensorflow web playground

A slightly less trivial example

- 2 linear+sigmoid hidden units no longer sufficient to do a good job.
- But we can find a reasonably good solution with 6 linear+sigmoid units.



Which properties do you want to feed in?

This is the output from one ***neuron****.*

NNs with one hidden layer are “universal approximators”

If we have enough hidden units

- We’ll just cover a quick visual intuition pump here.
- An early proof for sigmoid units was Cybenko, 1989:
  - Proof not required for exam!
  - Cybenko., G. (1989) **"Approximations by superpositions of sigmoidal functions"**, *Mathematics of Control, Signals, and Systems*, 2 (4), 303-314

- Nice interactive treatment in Michael Nielsen’s deep learning book here.

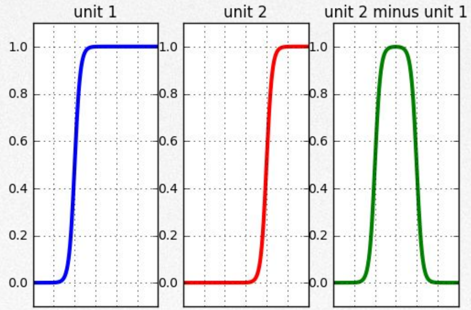
- Aside: Beyond scope of this class, but turns out that under certain additional assumptions we can consider neural nets with infinitely large hidden layers as Gaussian Processes and lots of sophisticated Bayesian techniques. Great resources here if you’d like to learn more about this.

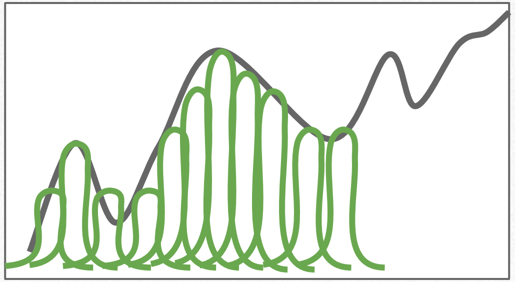
NNs with one hidden layer are “universal approximators”

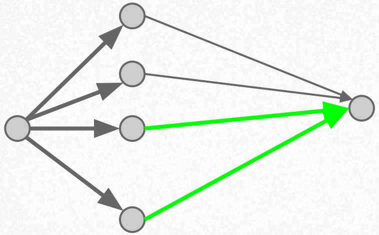
If we have enough hidden units

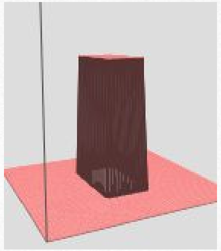
- “Visual proof”
- In 1D, can use pairs of offset sigmoid hidden units to form localised “bumps”.
- By translating and scaling these bumps, we can approximate arbitrary 1D functions.
- The more bumps we have, the steeper/narrower they can be, and the better our approximation.

- Similarly in higher-dimensions, except each high-dimensional bump requires larger groups of units.









Going deeper

aka “How I stopped worrying and learned to love the compute graph”

Section Overview

- Benefits of depth
  - 1 hidden layer gives universal approximation, but can require *many* hidden units.
  - Deeper networks can be much more powerful and efficient. [Intuitive account.]
  - Break complex functional mapping into series of smaller re-representation steps.
    - Perhaps: edges → junctions → parts → objects → scenes

- Compute graphs
  - Modern neural network applications can have very complex topologies.
  - Useful to think in terms of general compute graphs that we can train.
    - neural nets ↔ flexible, modular, non-linear functions
  - Object oriented approach.

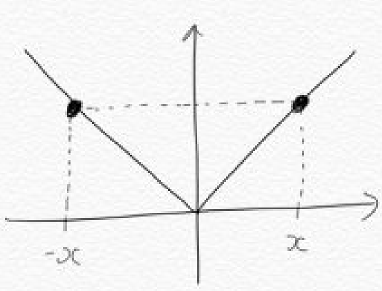
- Some examples of deep architectures.

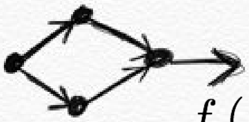
NOT FOR EXAM!!

Visual intuition pump for why adding depth helps

Exponentially more regions with depth

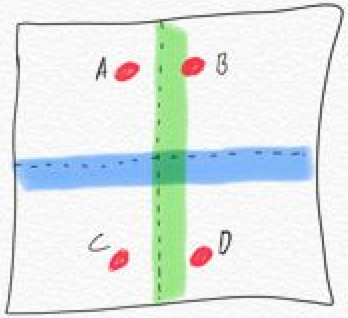
Figure from Montufar, Pascanu, Cho & Bengio, 2014

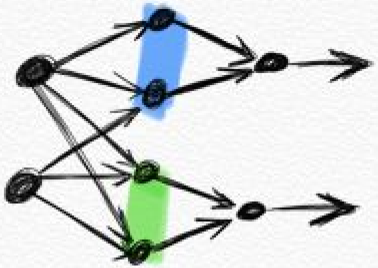




$$f\left(|x|\right) = f\left(|-x|\right)$$

- Can view layers of “full” rectifiers as folding space.
  - Multiple points in input map to same point in output.
  - Leads to multiple regions sharing same function mapping.
- Applied recursively (multiple layers) results in exponentially many regions.





$$f\left(|x|,|y|\right) = f\left(|-x|,|y|\right) = f\left(|x|,|-y|\right) = f\left(|-x|,|-y|\right)$$

NOT FOR EXAM!!

Visual intuition pump for why adding depth helps

Exponentially more regions with depth

Figure from Montufar, Pascanu, Cho & Bengio, 2014

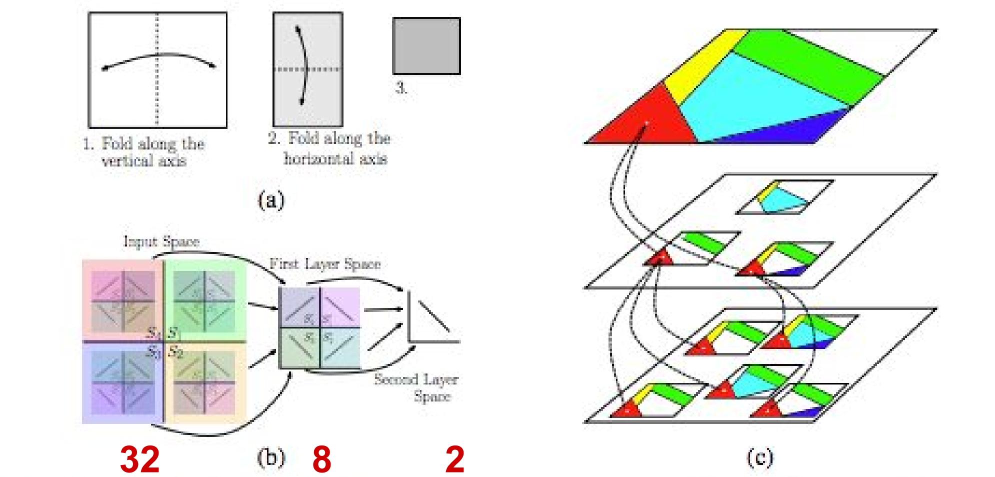


Figure 2: (a) Space folding of 2-D Euclidean space along the two axes. (b) An illustration of how the top-level partitioning (on the right) is replicated to the original input space (left). (c) Identification of regions across the layers of a deep model.

- Can view layers of "full" rectifiers as folding space.
  - Multiple points in input map to same point in output.
  - Leads to mutiple regions sharing same function mapping.
- Applied recursively (multiple layers) results in exponentially many regions.
- Paper goes on to show that the number of linear regions grows **exponentially** with depth and **polynomially** with the number of units per hidden layer.

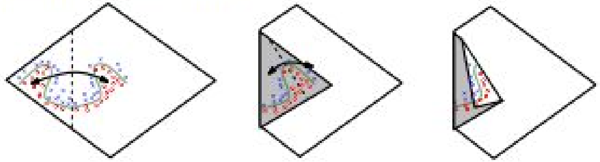
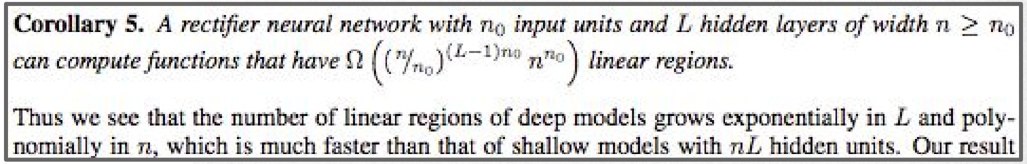


Figure 3: Space folding of 2-D space in a non-trivial way. Note how the folding can potentially identify symmetries in the boundary that it needs to learn.



Corollary 5. *A rectifier neural network with n_0 input units and L hidden layers of width $n \geq n_0$ can compute functions that have $\Omega\left(\left(n/n_0\right)^{(L-1)n_0} n^{n_0}\right)$ linear regions.*

Thus we see that the number of linear regions of deep models grows exponentially in L and polynomially in n, which is much faster than that of shallow models with nL hidden units. Our result

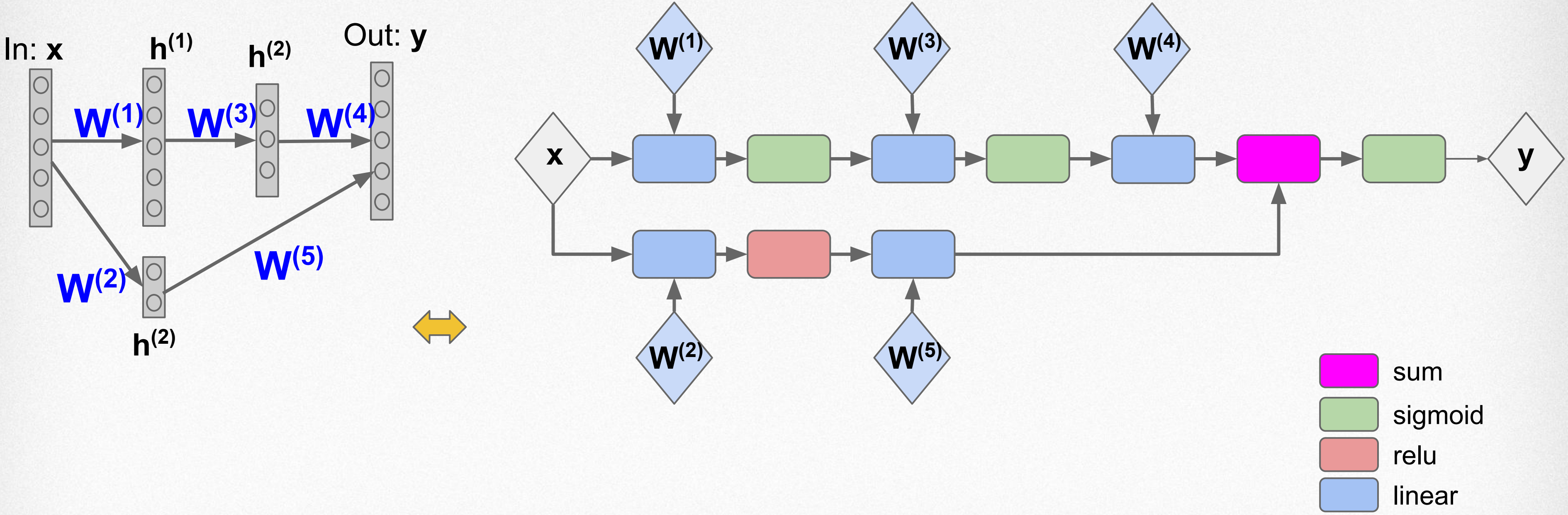
Dont worry if that last part went by a bit fast

Not for exam!

Neural networks viewed as compute graphs

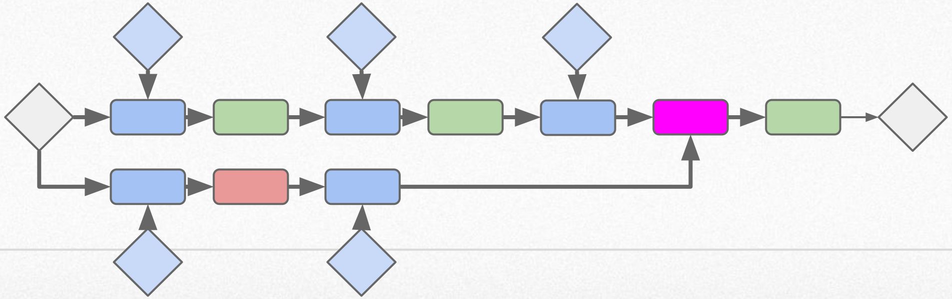
Different conventions for presenting network diagrams

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(1)}\mathbf{x}\right);\ \mathbf{h}^{(2)} = \mathrm{relu}\left(\mathbf{W}^{(2)}\mathbf{x}\right);\ \mathbf{h}^{(3)} = \sigma\left(\mathbf{W}^{(3)}\mathbf{h}^{(1)}\right);\ \mathbf{y} = \sigma\left(\mathbf{W}^{(4)}\mathbf{h}^{(3)} + \mathbf{W}^{(5)}\mathbf{h}^{(2)}\right);$$



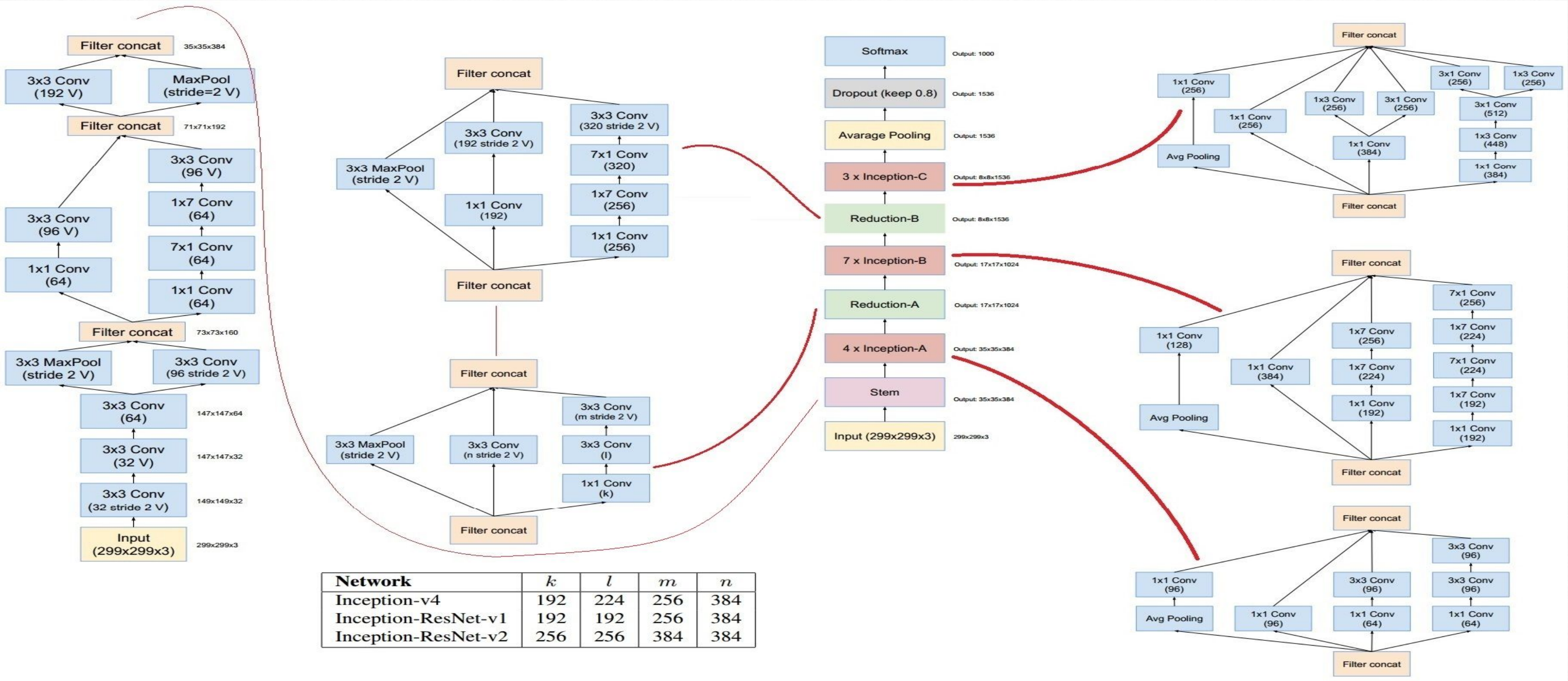
Neural networks viewed as compute graphs

- Library of computational building blocks (modules)
  - allows code reuse and lets us easily build many different models
  - analogous to lego :)
- Object oriented approach
  - Nice links between these building blocks and object oriented approaches.
  - Minimal set of API functions each object needs to provide is small.
    - `forward_pass()`
    - `backward_pass()`
    - `compute_gradients()`
  - Easy to add new modules if what we want is not supported.



Example: Inception V4

Images from: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning Szegedy et al, 2016

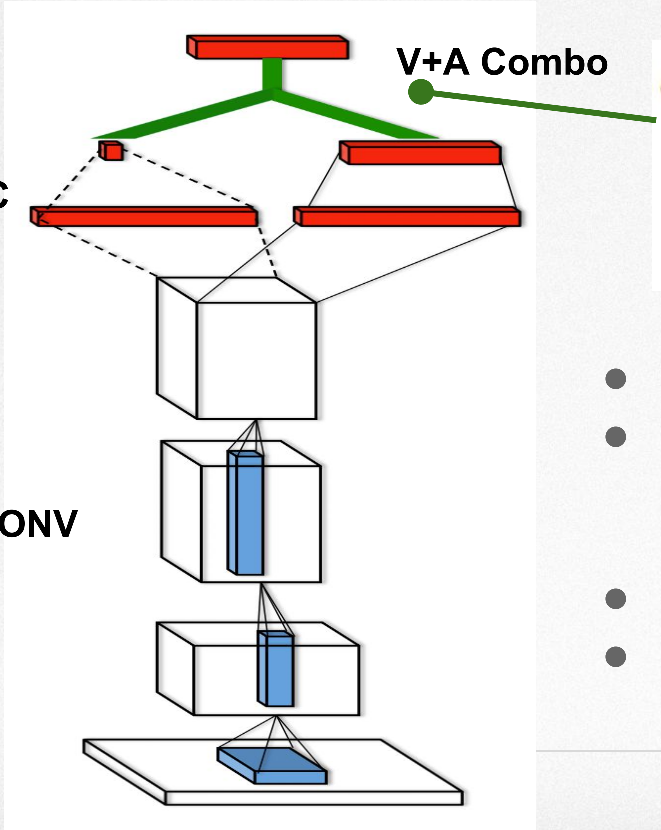


Network	k	l	m	n
Inception-v4	192	224	256	384
Inception-ResNet-v1	192	192	256	384
Inception-ResNet-v2	256	256	384	384

DeepMind

Example: Dueling nets for Deep RL

Figures from: Dueling Network Architectures for Deep Reinforcement Learning, Wang et al, 2015 (← inc Hado)



$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

- 3 conv layers (coming next lecture)
- Split into 2 branches of 2 fully-connected layers
  - One computes Value, V; the other a modified form of the Advantage, A (coming in later RL lectures).
- Final layer combines to give action-value function, Q.
- Significantly outperforms DQN on Atari suite.

Learning

Section Overview

- **Learning**:
  - Define a **loss function** with respect to our **data** and **model parameters**.
  - Optimization methods to find a set of model parameters that minimize the loss.
    - Typically radient descent (online / mini-batch / full batch)
    - Optimization lecture (James Martens) will cover this and other things in much more detail.

- Recap and notation setup for some useful matrix calculus.
- Recap simple gradient descent.
- Chain-rule and backprop on simple compute graphs.
- Automatic differentiation.
- Modular backprop.
  - Example: detailed derivation of module gradients for linear+softmax and multinomial loss.

Additional background reading : here, here, and here.

Linear algebra recap / notation

We'll use two types of object from matrix-calculus here

- **Gradient vector**:
  - Entries → partial derivatives of a *scalar* function wrt to *vector* arguments.
    - e.g. derivative of our loss with respect to parameters or inputs/outputs of a layer.
- **Jacobian matrix**:
  - Entries → partial derivatives of a *vector* function wrt to *vector* arguments.

Gradient

$$y = f(\mathbf{x})$$

$$f : \mathbb{R}^m \to \mathbb{R}$$

$$\frac{\partial y}{\partial \mathbf{x}} = \nabla^{(\mathbf{x})} y = \left[\frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_m}\right]$$

Jacobian

$$\mathbf{y} = f(\mathbf{x})$$

$$f : \mathbb{R}^m \to \mathbb{R}^n$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{J}^{(\mathbf{x})}(\mathbf{y}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

Optimization by gradient descent

Recap from Thore's lecture

- Full/mini batch gradient descent (entire/partial batch of data)

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla^{(\boldsymbol{\theta})} L\left(\boldsymbol{\theta}, \left\{\mathbf{x}^{(i)}, y^{(i)}\right\}_{i=1}^{m}\right)$$

- Online gradient descent (one datum at a time)

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla^{(\boldsymbol{\theta})} L\left(\boldsymbol{\theta}, \mathbf{x}, y\right)$$

- Choice of learning rate matters a lot! (Hyperparameter optimization.)
- Will see more sophisticated methods later in this course (James Martens)
  - Momentum, RMSProp, ADAM, etc
  - Nice overview here.

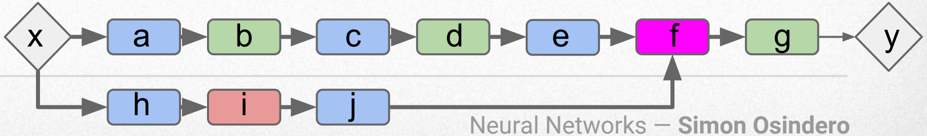
Chainrule, backprop and automatic differentiation

Basics

- Simple nested functions: $y = f\left(g\left(x\right)\right);\ \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x}$
- Multivariate: $y = f\left(g^{(1)}\left(x\right),\ldots,g^{(m)}\left(x\right)\right);\ \frac{\partial y}{\partial x} = \sum_{i=1}^{i=m}\frac{\partial f}{\partial g^{(i)}}\frac{\partial g^{(i)}}{\partial x}$
- Compute graphs→ product along single path; sum over all possible paths
  - Efficiency trick: sum/distribute all paths through a junction nodes as we traverse.
  - Traverse from inputs to outputs → forward mode AD
  - Traverse backwards from outputs to inputs → reverse mode AD → **backprop.**

$$y = g(f(j(i(h(x)))), e(d(c(b(a(x))))))$$

$$\frac{\partial y}{\partial x} = \frac{\partial a}{\partial x}\frac{\partial b}{\partial a}\frac{\partial c}{\partial b}\frac{\partial d}{\partial c}\frac{\partial e}{\partial d}\frac{\partial f}{\partial e}\frac{\partial g}{\partial f}\frac{\partial y}{\partial g} + \frac{\partial h}{\partial x}\frac{\partial i}{\partial h}\frac{\partial j}{\partial i}\frac{\partial f}{\partial j}\frac{\partial g}{\partial f}\frac{\partial y}{\partial g}$$



Chainrule, backprop and automatic differentiation

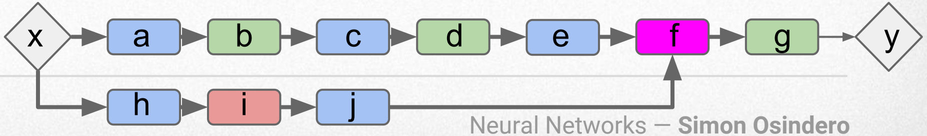
- **Forwards mode AD**:
  - Traverse graph from each input to output. $\frac{\partial a}{\partial x}, \frac{\partial b}{\partial x}, \frac{\partial c}{\partial x}, \dots, \frac{\partial y}{\partial x}$
  - Intermediate results provide derivative of each node wrt to the original input.

- **Backwards mode AD (i.e. backprop)**: $\frac{\partial y}{\partial g}, \frac{\partial y}{\partial f}, \frac{\partial y}{\partial e}, \dots, \frac{\partial y}{\partial x}$
  - Traverse graph from output to input.
  - Intermediate results provide derivative of the output with respect to each node.

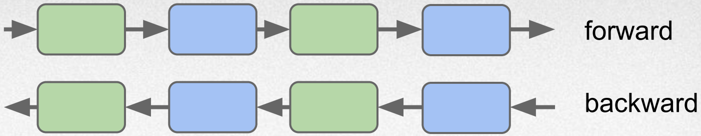
← need these to get param derivs wrt to scalar output (i.e. loss).

$$y = g(f(j(i(h(x))), e(d(c(b(a(x)))))))$$

$$\frac{\partial y}{\partial x} = \frac{\partial a}{\partial x}\frac{\partial b}{\partial a}\frac{\partial c}{\partial b}\frac{\partial d}{\partial c}\frac{\partial e}{\partial d}\frac{\partial f}{\partial e}\frac{\partial g}{\partial f}\frac{\partial y}{\partial g} + \frac{\partial h}{\partial x}\frac{\partial i}{\partial h}\frac{\partial j}{\partial i}\frac{\partial f}{\partial j}\frac{\partial g}{\partial f}\frac{\partial y}{\partial g}$$



Automatic differentiation

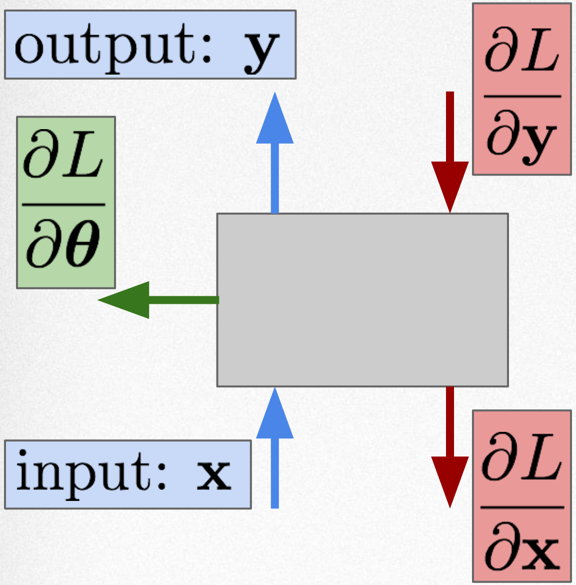


Summary: Backprop ↔ Reverse mode AD

- Backwards pass:
  - compute derivative of the loss wrt inputs of a module, given the derivatives of the loss wrt outputs of that module.
- Part of module API when building software. (fprop/bprop)
- Can apply this to the *full* compute graph (loops, conditionals, etc) → just backtrack through the forward execution path.
  - Need to store variables during forward path (memory)
    - But there are clever tricks if needed
      - Memory-Efficient Backpropagation Through Time
  - Packages like TF take care of storage and reverse traversal automagically.
- Also a nice blog post on this topic.

Modular backprop with vector input/outputs

Minimal set of module methods: forward-pass, backward-pass, and parameter gradients



output: $\mathbf{y}$
$\frac{\partial L}{\partial \boldsymbol{\theta}}$
$\frac{\partial L}{\partial \mathbf{y}}$
input: $\mathbf{x}$
$\frac{\partial L}{\partial \mathbf{x}}$

- Forward-pass

```
def forward_pass(x):
    # Compute output given input
    return y
```

$$\mathbf{y} = f\left(\mathbf{x}\right)$$

- Backward-pass

```
def backward_pass(dL_dy):
    # Compute gradient of loss with respect to inputs,
    # given gradient of loss with respect to outputs
    return dL_dx
```

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{J} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \left(\mathbf{J}^{(\mathbf{x})}\left(\mathbf{y}\right)\right)$$

- Parameter gradients

```
def param_gradients(dL_dy):
    # Compute gradient of loss with respect to parameters
    # given gradient of loss with respect to outputs
    return dL_dtheta
```

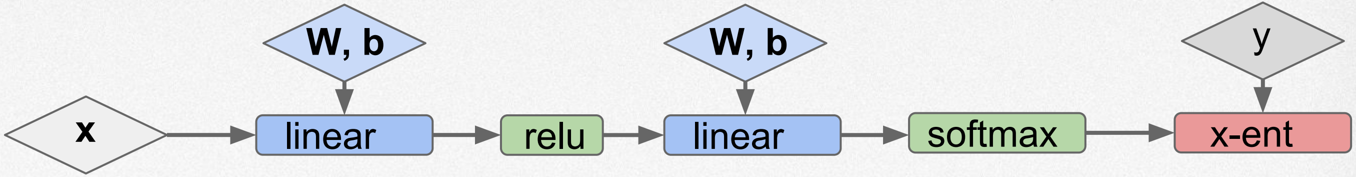
$$\frac{\partial L}{\partial \theta_i} = \sum_{j=1}^{J} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial \theta_i}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \frac{\partial L}{\partial \mathbf{y}} \left(\mathbf{J}^{(\boldsymbol{\theta})}\left(\mathbf{y}\right)\right)$$

- Can express cleanly with gradient x Jacobian.
- But, usually not efficient to explicitly form matrix and multiply. Often best to derive directly and/or use index notation and look for simplifications.

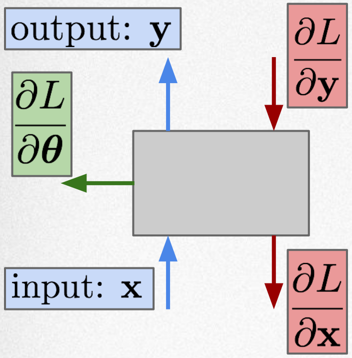
Modular backprop: linear + softmax + cross-entropy loss

Worked example



- For each module, we just need to write down expressions for:
  - **forward_pass**:
    - outputs given inputs
  - **backward_pass**:
    - derivatives of loss wrt inputs given derivatives of loss wrt outputs
  - **param_gradients**:
    - derivatives of loss wrt parameters given derivatives of loss wrt outputs
- Chain together forward passes, then chain together backward passes and gradient computations.
- Apply gradients & iterate.

linear module


output: $\mathbf{y}$
$\frac{\partial L}{\partial \mathbf{y}}$
$\frac{\partial L}{\partial \boldsymbol{\theta}}$
input: $\mathbf{x}$
$\frac{\partial L}{\partial \mathbf{x}}$

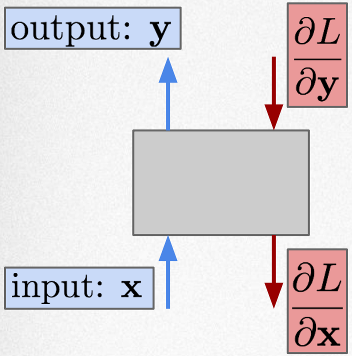
- forward_pass: $\boxed{\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}}$ $\quad y_n = \sum_m W_{nm} x_m + b_n$
- Jacobian elements: $\frac{\partial y_i}{\partial x_j} = W_{ij} \;\; ; \;\; \frac{\partial y_i}{\partial b_j} = \delta_{ij} \;\; ; \;\; \frac{\partial y_i}{\partial W_{jk}} = x_k \delta_{ij}$
- backward_pass: $\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_j} = \sum_i \frac{\partial L}{\partial y_i} W_{ij}$ $\quad \boxed{\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}}$
- param_gradients:

$$\frac{\partial L}{\partial W_{jk}} = \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial W_{jk}} = \sum_i \frac{\partial L}{\partial y_i} x_k \delta_{ij} = \frac{\partial L}{\partial y_j} x_k$$

$$\boxed{\frac{\partial L}{\partial \mathbf{W}} = \left(\frac{\partial L}{\partial \mathbf{y}}\right)^T \mathbf{x}^T}$$

$$\boxed{\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{y}}}$$

relu module



output: $\mathbf{y}$

$\frac{\partial L}{\partial \mathbf{y}}$

input: $\mathbf{x}$

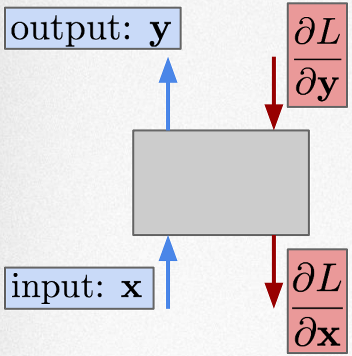
$\frac{\partial L}{\partial \mathbf{x}}$

- forward_pass: $\boxed{\mathbf{y} = \mathrm{relu}(\mathbf{x})}$ $\qquad y_i = \max(0, x_i)$

- backward_pass:

$$\frac{\partial L}{\partial x_i} = (y_i > 0)$$

softmax module



output: $\mathbf{y}$
$\frac{\partial L}{\partial \mathbf{y}}$
input: $\mathbf{x}$
$\frac{\partial L}{\partial \mathbf{x}}$

- forward_pass: $\mathbf{y} = \text{softmax}\left(\mathbf{x}\right)$

$$y_n = \frac{e^{x_n}}{\sum_m e^{x_m}}$$

- Jacobian elements:

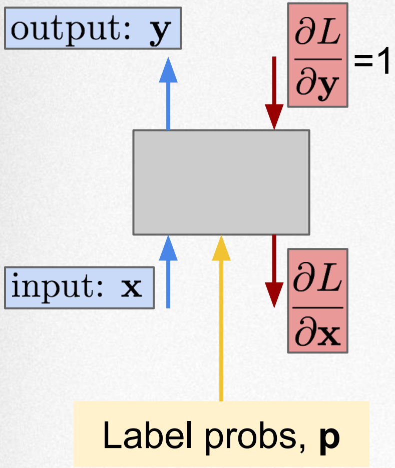
$$\frac{\partial y_i}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{e^{x_i}}{\sum_m e^{x_m}}\right) = \frac{\delta_{ij} e^{x_i}}{\sum_m e^{x_m}} - \frac{e^{x_i} e^{x_j}}{\left(\sum_m e^{x_m}\right)^2} = y_i\left(\delta_{ij} - y_j\right)$$

- backward_pass:
  - derivation in assignment/check my maths :) ...

$$\frac{\partial L}{\partial \mathbf{x}} = \mathbf{s} - \sum_i s_i; \text{ where } s_i = \frac{\partial L}{\partial y_i} y_i$$

cross-entropy loss module

With softmax-out as inputs



output: $\mathbf{y}$
$\frac{\partial L}{\partial \mathbf{y}}$ =1
input: $\mathbf{x}$
$\frac{\partial L}{\partial \mathbf{x}}$
Label probs, **p**

- forward_pass: $\boxed{y = L = \mathrm{Xent}\left(\mathbf{p}, \mathbf{x}\right)}$

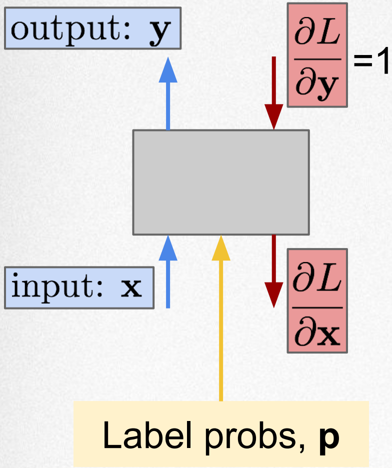
$$y = -\sum_{i}^{\mathrm{Classes:C}} p_i \log x_i$$

- backward_pass: $\frac{\partial L}{\partial x_i} = -\frac{p_i}{x_i}$

Compound module

cross-entropy loss module

With **logits** (softmax-inputs) as inputs



output: $\mathbf{y}$
$\frac{\partial L}{\partial \mathbf{y}}$ =1
input: $\mathbf{x}$
$\frac{\partial L}{\partial \mathbf{x}}$
Label probs, **p**

- forward_pass: $\boxed{y = L = \text{XentLogits}\left(\mathbf{p}, \mathbf{x}\right)}$

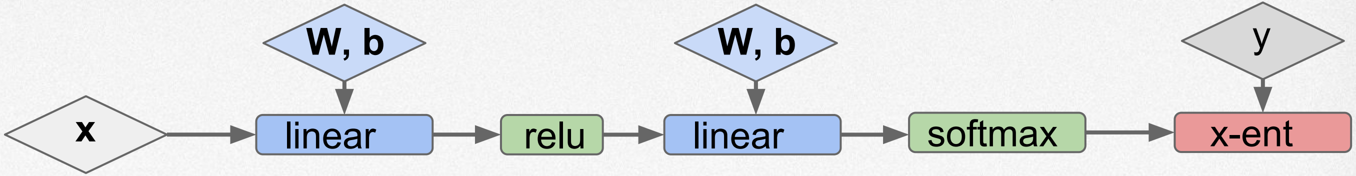
$$y = -\sum_{i}^{\text{Classes:C}} p_i \log\left(\frac{e^{x_i}}{\sum_m e^{x_m}}\right)$$

- backward_pass:
  - derivation in assignment ...

$$\frac{\partial L}{\partial \mathbf{x}} = \mathbf{y} - \mathbf{p}$$

Modular backprop: linear + softmax + cross-entropy loss

Putting it together



- Online SGD with our modules:
    - for i = 1… num_iterations:
        - Get an input & label (x,y)
        - Chain together forward_passes to get loss
        - Chain together backward passes and parameter gradient computations.
        - Apply gradients updates
- Adding additional hidden layers or changing the topology is easy
    - E.g. extra hidden layer →
        - just instantiate an extra linear module and an extra relu module
        - change the call sequence slightly

Basic module zoo

Brief tour of some modules commonly used in neural networks

Section Overview

List far from exhaustive + modular bp makes it trivial to make arbitrary modules of our own

- Linear modules with parameters
  - Linear
  - ~~Convolutional / Deconvolutional~~ [*covered in next lecture → Karen Simonyan*]
- Basic elementwise ops
    - add, multiply,
- Group ops
    - softmax, max, sum
- Elementwise non-linear functions (activation functions)
    - relu, sigmoid, tanh, leaky relu
- Basic loss functions
  - Cross-entropy / negative log-likelihood
  - Squared error

See here for a summary of more functions.
Also, the lecture by Raia Hadsell later in this course.

Elementwise ops

add / multiply

- Add: fprop: $\mathbf{y} = \mathbf{a} + \mathbf{b}$; bprop: $\dfrac{\partial L}{\partial \mathbf{a}} = \dfrac{\partial L}{\partial \mathbf{y}}$
- Multiply: fprop: $\mathbf{y} = \mathbf{a} \odot \mathbf{b}$; bprop: $\dfrac{\partial L}{\partial \mathbf{a}} = \mathbf{b} \odot \dfrac{\partial L}{\partial \mathbf{y}}$

Groupwise ops

sum / max / switch

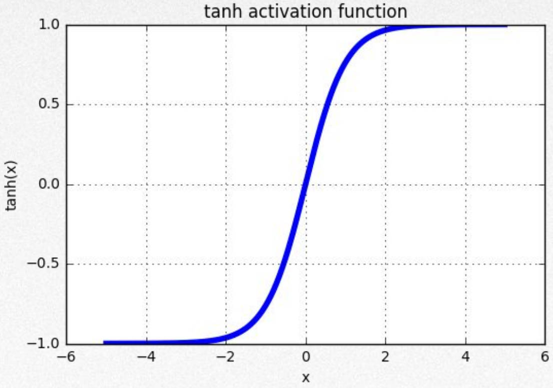
- Sum: $\text{fprop: } \mathbf{y} = \sum x_i \text{ ; bprop: } \frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial y}\mathbf{1}^T$
- Max: $\text{fprop: } \mathbf{y} = \max_i\{x_i\} \text{ ; bprop: } \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y} \text{ if } i \text{ was maximal, 0 otherwise}$
- Switch/Conditional:
  - Denote active branch by one-hot vector, **s**.

$$\text{fprop: } \mathbf{y} = \mathbf{s} \odot \mathbf{x} \text{ ; bprop: } \frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \odot \mathbf{s}^T$$

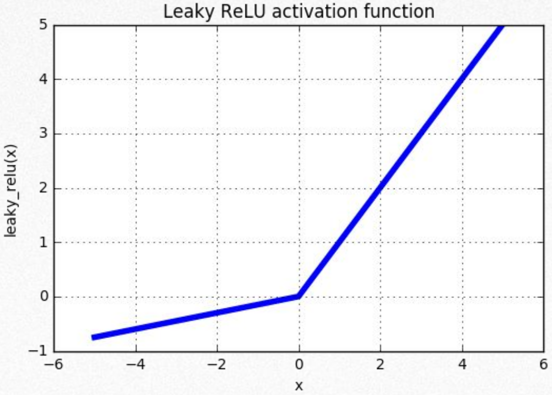
Activation functions

tanh / leaky relu

- tanh
  - Scaled and shifted version of sigmoid



- Leaky relu
  - Overcomes possible issue with regular relu that gradients are zero when inputs are negative.



Loss ops

- Squared error
  - E.g. for regression problems

$$\text{fprop:}\;\; L = y = \left\|\mathbf{t} - \mathbf{x}\right\|^2 = \sum_j \left(t_j - x_j\right)^2 \;\; ; \text{bprop:}\;\; \frac{\partial L}{\partial \mathbf{x}} = -2\left(\mathbf{t} - \mathbf{x}\right)^T$$

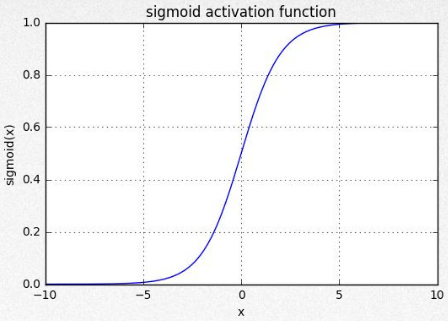
Practical issues

Section Overview

A few (high-level) practical tips

- Overfitting & regularization
    - Weight decay
    - Dropout
    - Data augmentation
- The importance of good initialization
    - Scaling schemes
    - Batch-norm [More to follow in Karen’s lecture.]
- Hyper-parameter optimization & architecture design
    - Random search over sensible range is usually preferable to grid search when d > 1.
    - Ongoing research to improve here. (Bayesian / evolutionary / RL to help guide search)
- A few tips for diagnosing/debugging
    - Check for dead units
    - Be aware of vanishing/exploding gradients. [More to follow in Karen, Oriol, and James’ lectures.]
    - Try to overfit on a very small subset of data or a very simple version of your task.
- Additional resources

Overfitting & Regularization



- Early stopping
- Weight decay
  - Keeps stops weights from getting to big.
  - Intuition: sigmoids/tanh/softmax are closer to their “linear region”
  - Doesn’t really affect relu
- Noise addition → dropout
  - Add noise to activities during training.
    - Prevents over-reliance on very specific conjunctions/ excessive co-coadaptation of features
    - Can also be viewed as a cheap way of doing ensembling
  - **Dropout:**
    - Training: randomly set fraction of activities in a layer to zero!!
    - Testing: use all units, but scale by dropout fraction **or** do multiple passes with different random fractions set to zero.

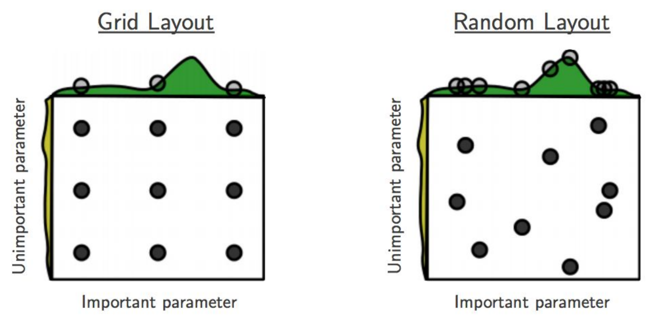
The importance of good initialization

- Initial weight settings should typically be small, but distribution/how small does matter.
    - Affects size of gradients and how easily information can propagate.
        - Too big → explode. Too small → vanish.
- Various heuristics, e.g. "Xavier initialization"
- **Batch-norm**:
    - Has been extremely important → helps a lot.
    - Scale (and offset) empirically.
        - Look at inputs to non-linearities and apply a scale and offset to make sure these are appropriate.
            - E.g. based on data statistic, add a layer after each linear to set the mean of each element to zero and the variance to 1. Apply this through the network.
        - Turns out this is good not just for initialization, but during training too.
            - We add trainable correction factors so we can undo the scaling if warranted.
            - Also acts as a regularize (intra-batch variability of normalization → noise)
    - More about this in Karen Simonyan's lecture on convnets.
            - Also: original paper here, and a video lecture by one of the authors here.

Hyperparameter search

How to pick good values for learning rates, dropout-fractions, weight-decay, etc

- Basically → just try many combinations, evaluate on held out data, and pick the best.
  - With many hyperparameters, search space can be huge. (Curse of dimensionality.)



In many situations, random search may be preferable to grid search.
As motivated in Bergstra & Bengio, here from which the figure is taken.

- Can also try to model space of hyperparams (in terms of solution quality) and do smarter search. (e.g. Snoek, Larochelle, Adams 2012). Alternatively, evolutionary algorithms, such as CMA-ES.
- Research frontier: RL algorithms that can dynamically adjust hyperparams → meta-learning. (e.g. Andrychowiz et al.)

Simple debugging and performance diagnostics

Lots more to say here, but fuzzy on details → getting the best performance is something of a dark art!

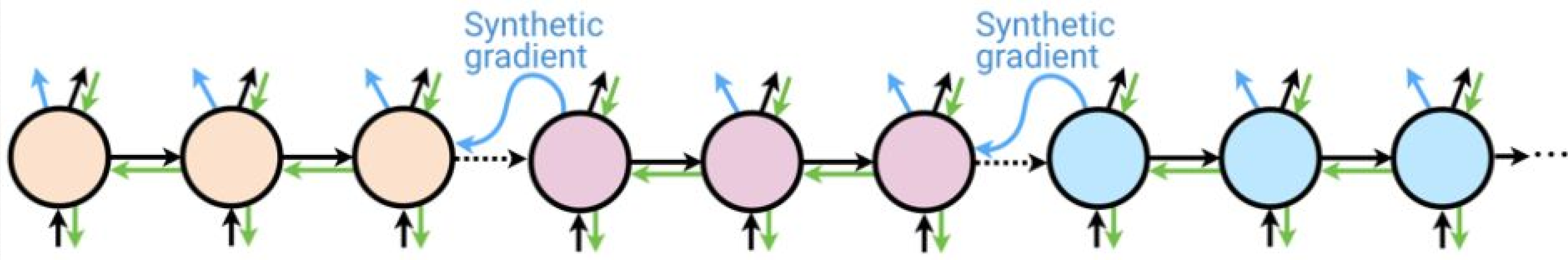
- Check for “dead units”
  - Histogram/visualize activities over large minibatch
- Be aware of vanishing/exploding gradients. [More to follow in Karen, Oriol, and James’ lectures.]
  - Histogram/visualize gradient updates over large minibatch
  - Rough rule of thumb: updates ~ 10^{-3} x parameter scale
- Try to overfit on a very small subset of data or a very simple version of your task.
  - Your model usually ought to be able to get training zero error on small datasets or simple versions of your task.

Research topic

Decoupled Neural Interfaces using Synthetic Gradients

NOT FOR EXAM!!

Decoupled Neural Interfaces Using Synthetic Gradients



- Can we build a local *model* to predict what back-propagated gradients will be, just based state of current module? (Turns out: Yes!)
- Why might this be useful?
    - In extremely large/complex compute graphs, sometimes need to do *lots* of computation before we can perform backprop.
    - We might benefit (learning speed-up) by being able to update parameters before the true gradients are available.
- DeepMind colleague, Max Jaderberg, made a great blog post here.

Next lecture...

Convolutional Neural Nets & Imagenet Models - Karen Simonyan

Appendix

[Optional] suggested readings and references

Bibliography

- Many of the slides in this presentation contain contextually relevant hyperlinks out to related content. (Reading is not required, but recommended if you're particularly interested in that topic.)

- Some useful (and free!) online textbooks
    - Deep Learning - Goodfellow, Bengion & Courville
    - Neural Networks & Deep Learning - Michael Nielsen
    - Information Theory, Inference, and Learning Algorithms - David Mackay