Machine learning - HT 2016
8. Neural Networks

Varun Kanade

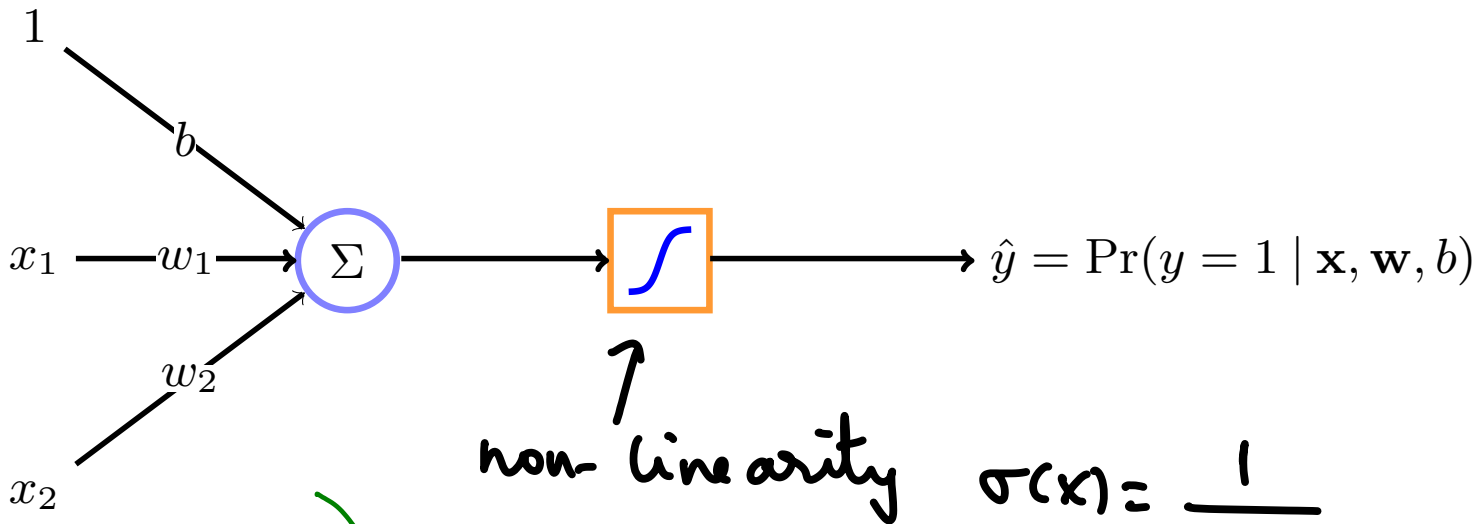
University of Oxford
February 19, 2016

Outline

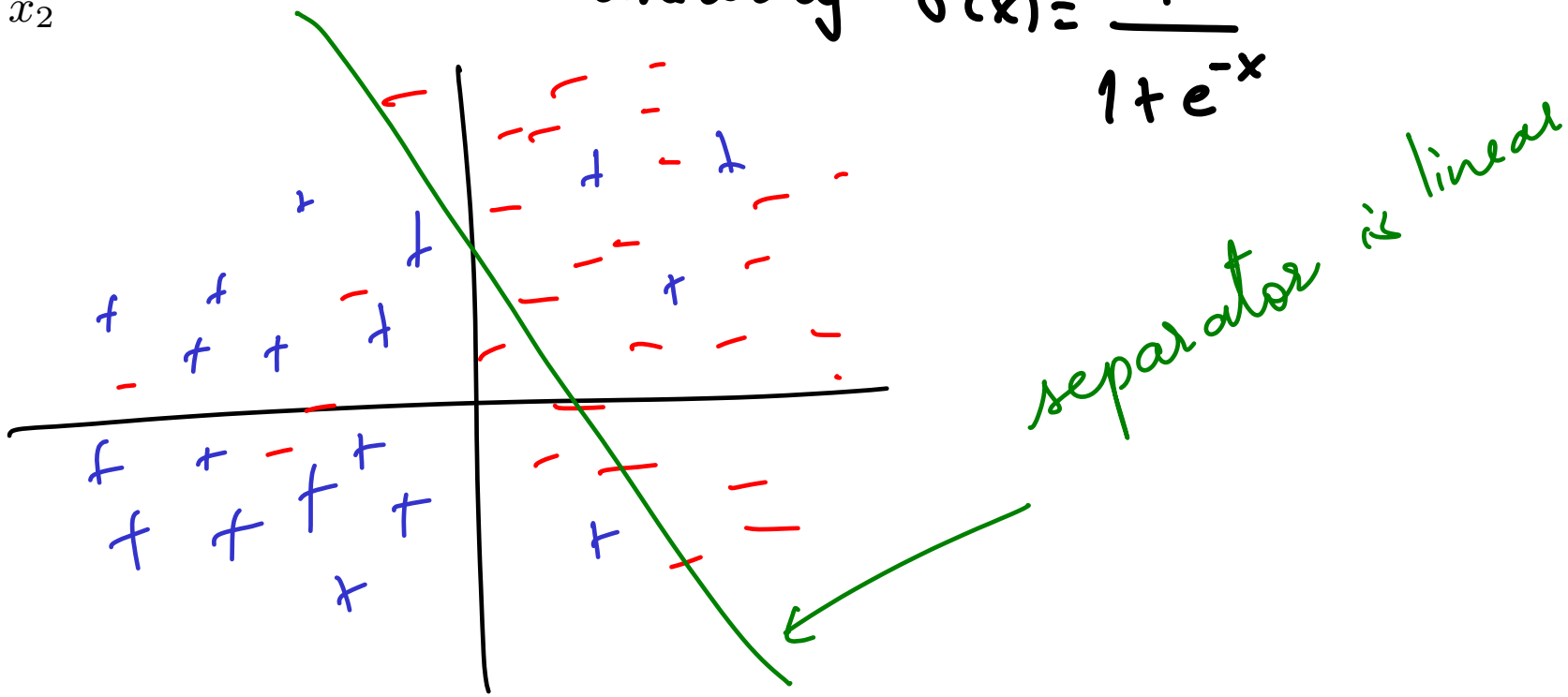
Today, we'll study feedforward neural networks

- Multi-layer perceptrons
- Application to classification or regression settings
- Backpropagation to compute gradients
- Finally understand what `model:forward` and `model:backward` is doing

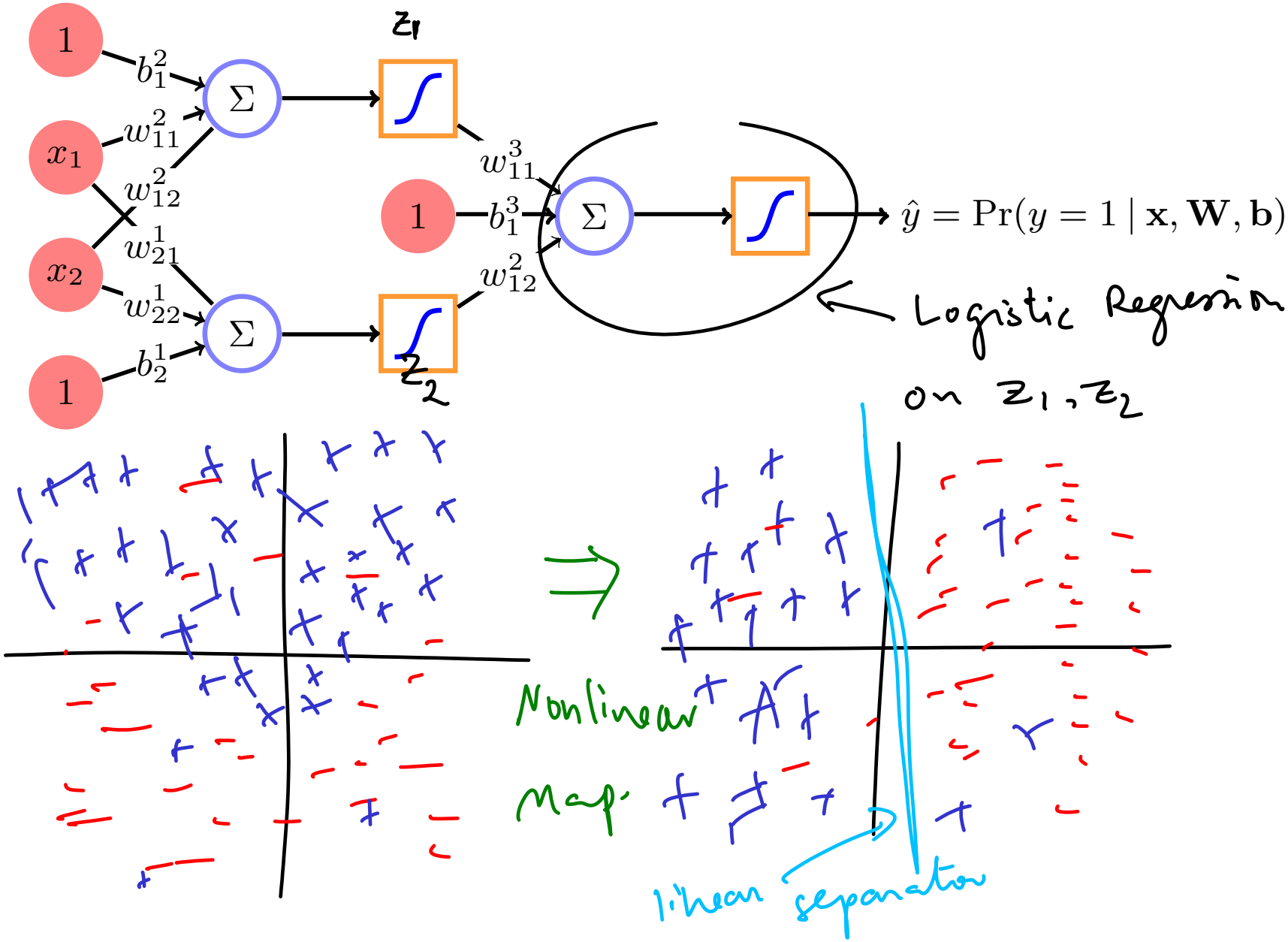
Artificial Neuron : Logistic Regression



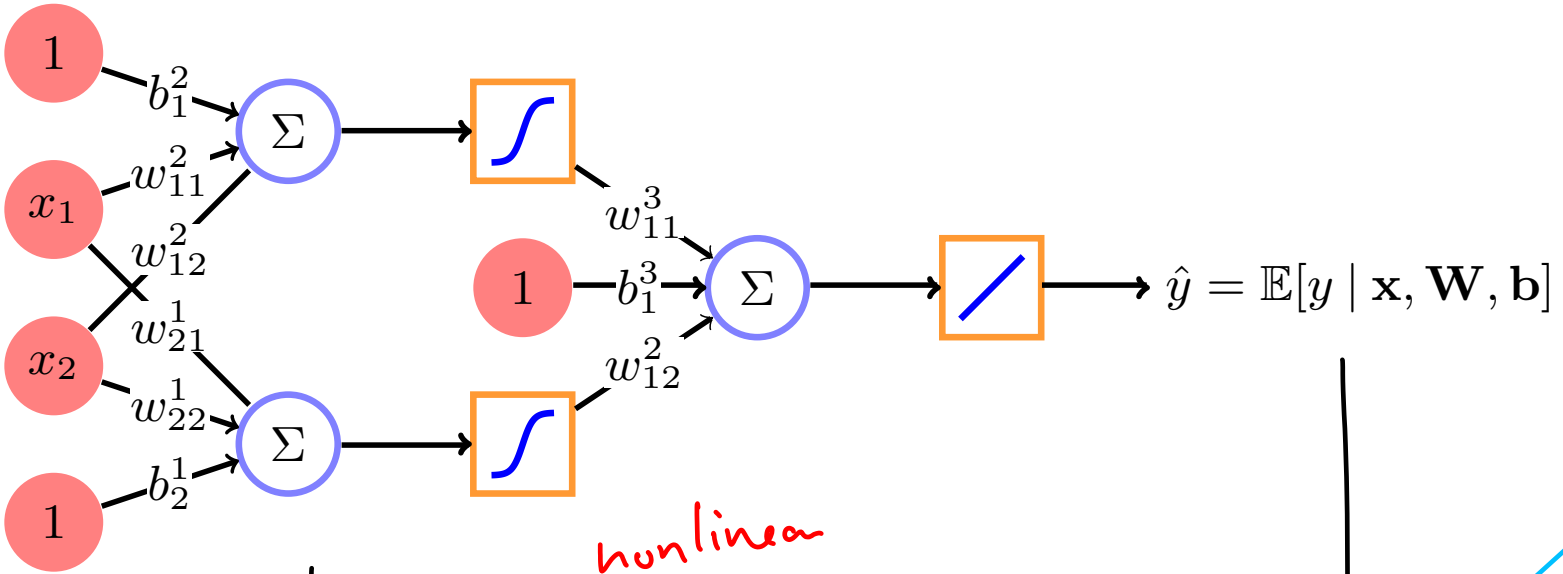
non-linearity $\sigma(x) = \frac{1}{1+e^{-x}}$

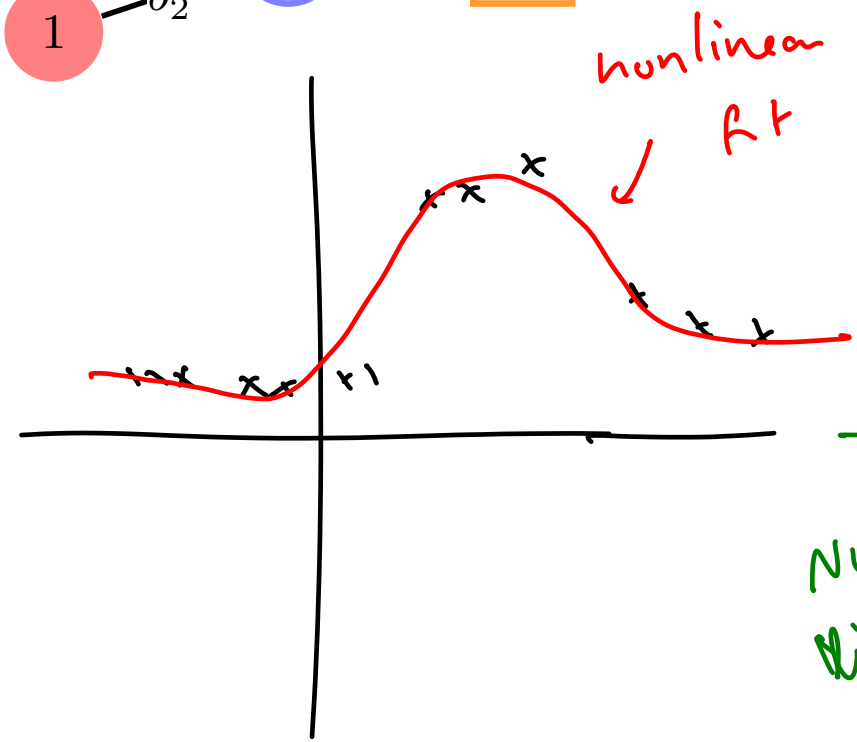


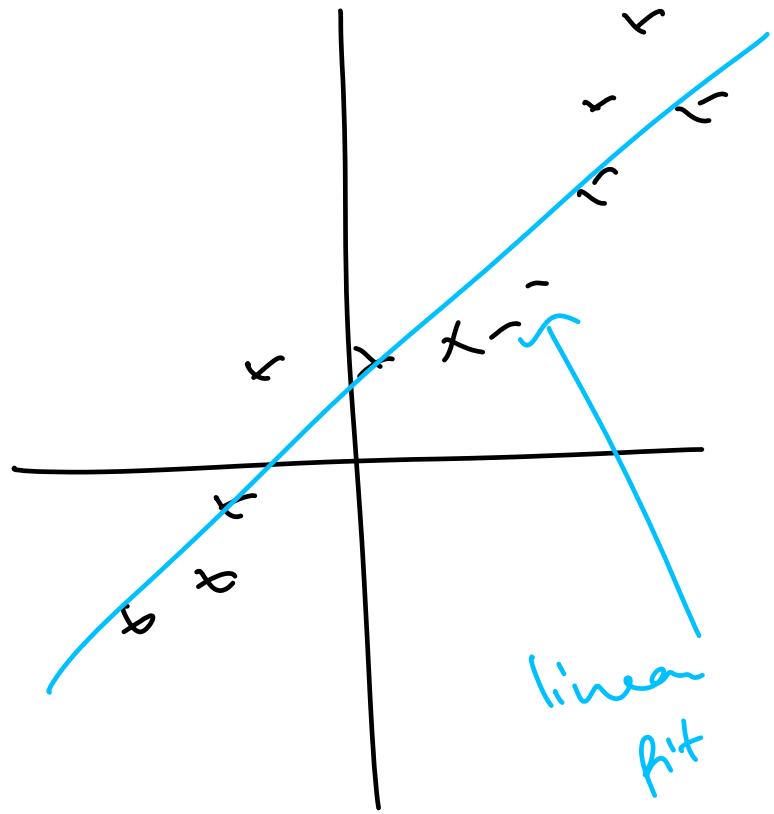
Multilayer Perceptron (MLP) : Classification



Multilayer Perceptron (MLP) : Regression

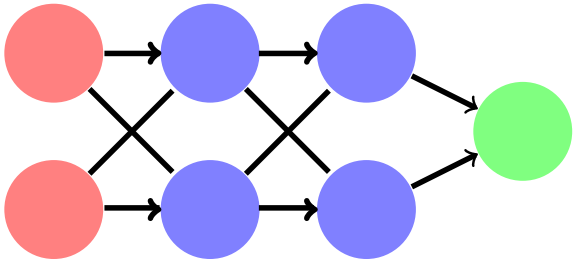






Non linear map

A Simple Example



$$z_1^4 = w_{11}^4 \sigma(z_1^3) + w_{12}^4 \sigma(z_2^3) + b_1^4$$

$$z_1^3 = w_{11}^3 \sigma(z_1^2) + w_{12}^3 \sigma(z_2^2) + b_1^3$$

$$z_2^3 = w_{21}^3 \sigma(z_1^2) + w_{22}^3 \sigma(z_2^2) + b_2^3$$

$$z_1^2 = w_{11}^2 x_1 + w_{12}^2 x_2 + b_1^2$$

$$z_2^2 = w_{21}^2 x_1 + w_{22}^2 x_2 + b_2^2$$

$$\hat{y} = \sigma\Big(w_{11}^4 \sigma\left(w_{11}^3 \sigma(w_{11}^2 x_1 + w_{12}^2 x_2 + b_1^2) + w_{12}^3 \sigma(w_{21}^2 x_1 + w_{22}^2 x_2 + b_2^2) + b_1^3\right)$$

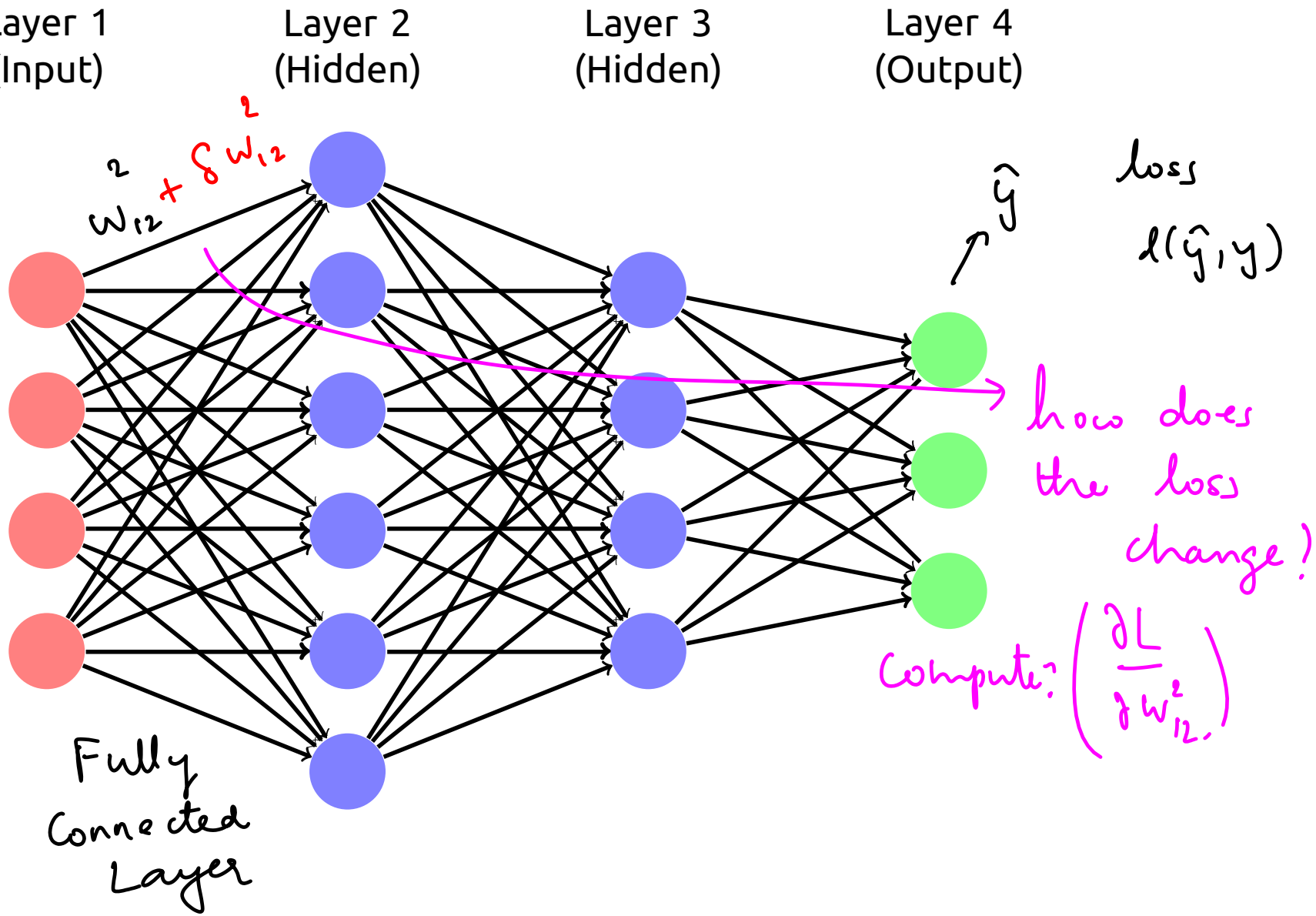
$$+ w_{12}^4 \sigma\left(w_{21}^3 \sigma(w_{11}^2 x_1 + w_{12}^2 x_2 + b_1^2) + w_{22}^3 \sigma(w_{21}^2 x_1 + w_{22}^2 x_2 + b_2^2) + b_1^3\right) + b_1^4 \Big)$$

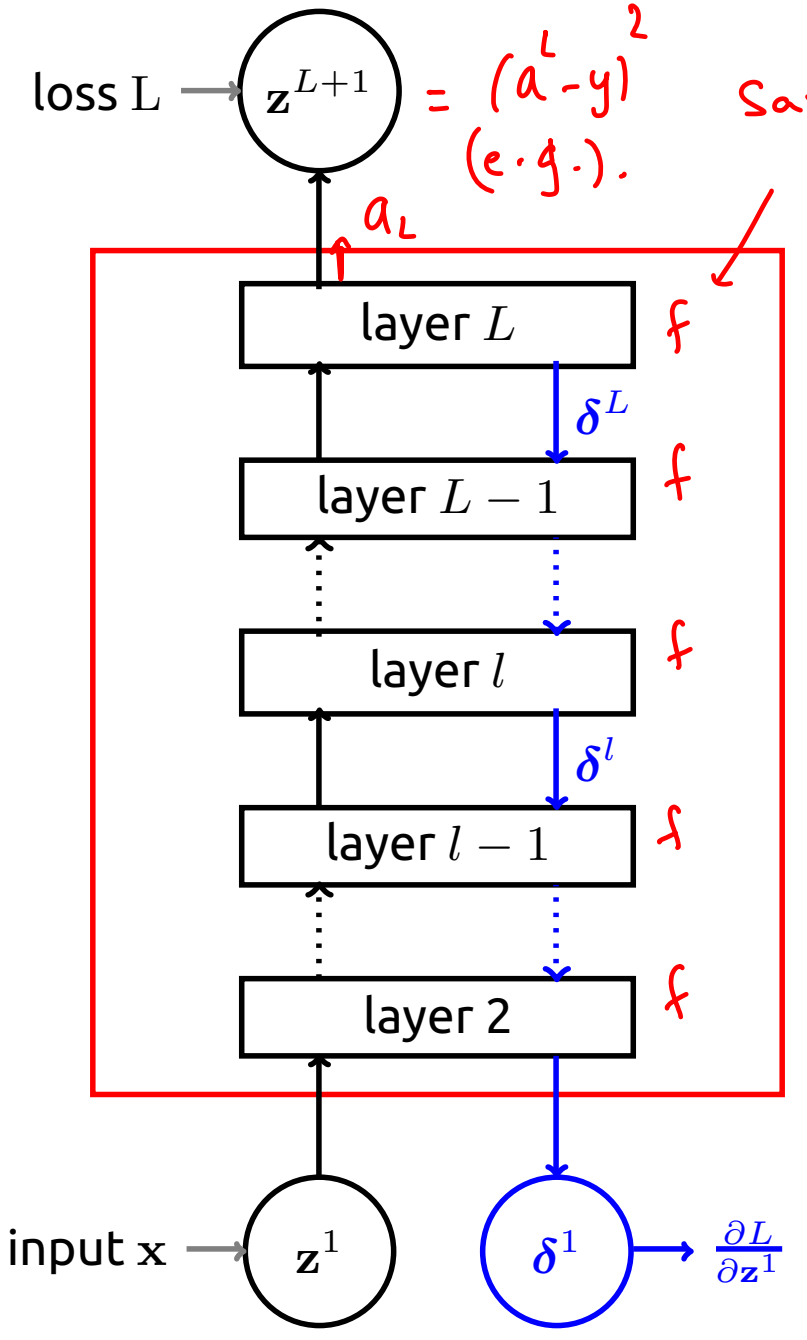
$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

$$\frac{\partial L}{\partial w_{12}^2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_{12}^2} = (\hat{y} - y) \cdot \Big[\sigma'(z_1^4) \cdot \Big[w_{11}^4 \cdot \sigma'(z_1^3) \cdot w_{11}^3 \, \sigma'(z_1^2) \cdot x_1 + w_{12}^4 \cdot \sigma'(z_2^3) \cdot w_{21}^3 \, \sigma'(z_1^2) \cdot x_1 \Big] \Big]$$

Want to exploit common occurrences!

Feedforward Neural Networks





$= (a^L - y)^2$ (e.g.).

Same non-linearity applied at every level.

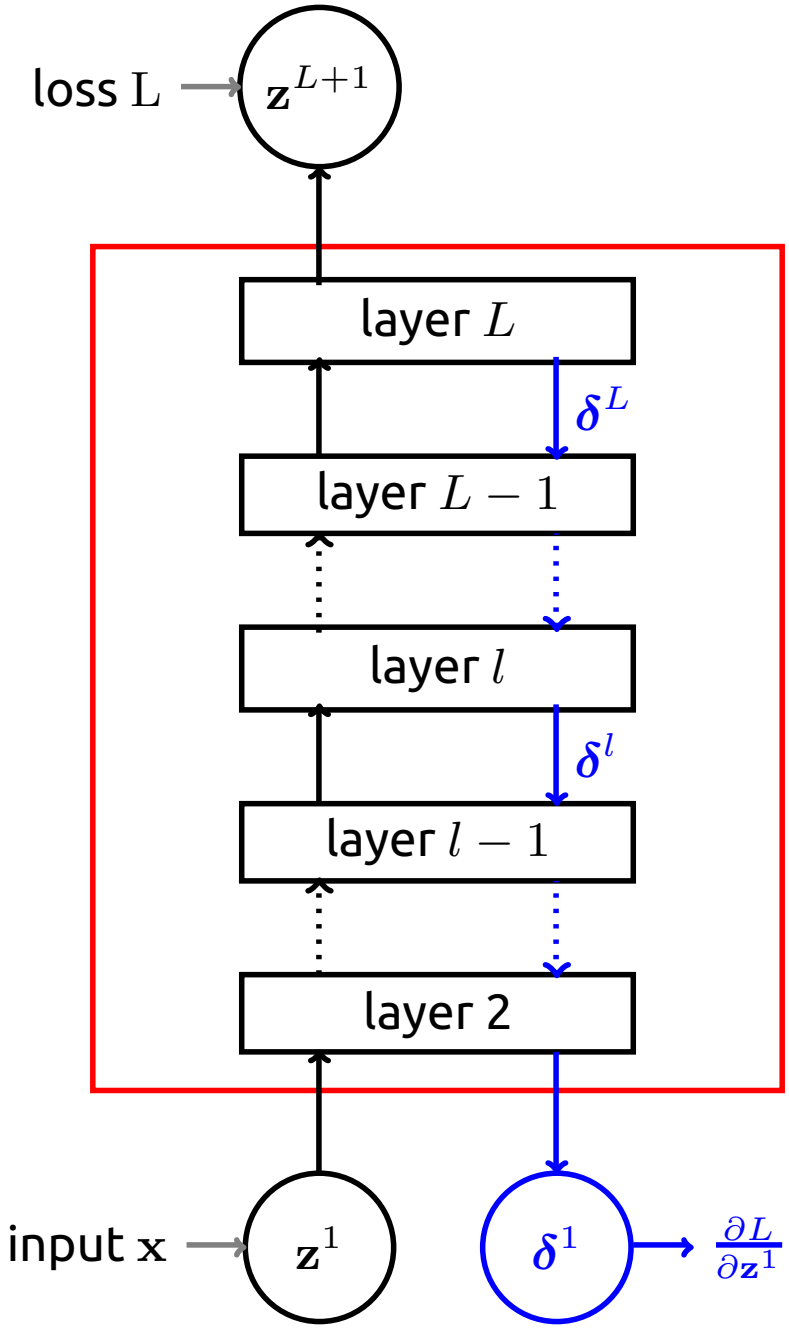
```
model = nn.Sequential()
model:add(nn.Linear(n_in, n_h1));
model:add(nn.Sigmoid());
...
...
model:add(nn.Linear(n_h_L-1, n_out));
model:Sigmoid();
criterion = nn.MSECriterion();

model:forward(x)
criterion:forward(model.output, y)

dL_do = criterion:backward(model.output,y)
model:backward(x, dL_do)
```

computes all the δ^l

$\frac{\partial L}{\partial a^L}$



Forward Equations

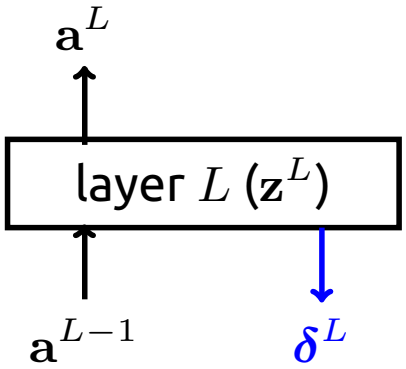
(1) $\mathbf{z}^1 = \mathbf{x}$ (input)

(2) $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$

(3) $\mathbf{a}^l = f(\mathbf{z}^l)$

(4) $\mathrm{L}(\mathbf{a}^L, y)$

Output Layer



$$\mathbf{z}^L = \mathbf{W}^L \mathbf{a}^{L-1} + \mathbf{b}^L$$

$$\mathbf{a}^L = f(\mathbf{z}^L)$$

Loss: $\mathrm{L}(y, \mathbf{a}^L) = \texttt{criterion : forward}(\mathbf{a}^{\mathrm{L}}, \mathrm{y})$

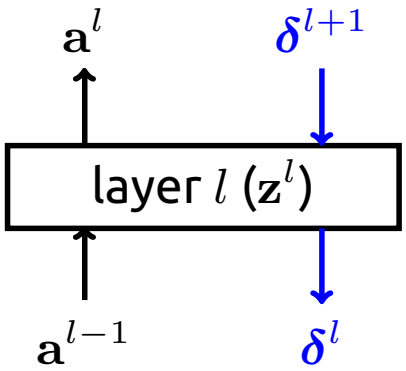
$$\boldsymbol{\delta}^L = \frac{\partial L}{\partial \mathbf{z}^L} = \frac{\partial L}{\partial \mathbf{a}^L} \odot f'(\mathbf{z}^L)$$

Hadamard product (element-wise vector mult)

$$\frac{\partial L}{\partial z_i^L} = \frac{\partial L}{\partial a_i^L} \cdot f'(z_i^L)$$

$$\frac{\partial L}{\partial z^L} = \frac{\partial L}{\partial a^L} \odot f'(z^L)$$

Back Propagation



$\mathbf{a}^l$ (the inputs into layer $l+1$)

$\mathbf{z}^{l+1} = \mathbf{W}^{l+1}\mathbf{a}^l + \mathbf{b}^{l+1}$ ($w^{l+1}_{j,k}$ weight on connection from k^{th} unit in layer l to j^{th} unit in layer $l+1$)

$\mathbf{a}^l = f(\mathbf{z}^l)$ (f is a non-linearity)

$\boldsymbol{\delta}^{l+1} = \frac{\partial \mathrm{L}}{\partial \mathbf{z}^{l+1}}$

$\boldsymbol{\delta}^l = \frac{\partial \mathrm{L}}{\partial \mathbf{z}^l} = \left(\frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{z}^l}\right)^T \cdot \frac{\partial \mathrm{L}}{\partial \mathbf{z}^{l+1}}$ ← Jacobian

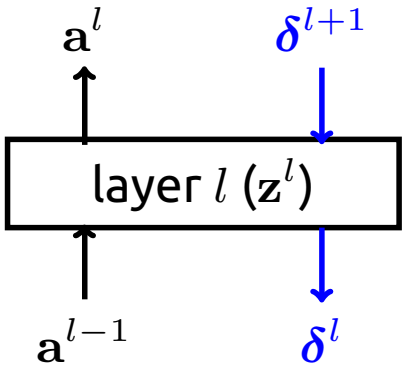
$$\frac{\partial z_i^{l+1}}{\partial z_j^l} = f'(z_j^l) \cdot w_{ij}^{l+1}$$

$$\frac{\partial z^{l+1}}{\partial z^l} = W^{l+1} \cdot \mathrm{Diag}(f'(a_j^l))$$

$$\delta^l = \mathrm{Diag}(f'(a_j^l)) \cdot (W^{l+1})^T \delta^{l+1}$$

$$= f'(a^l) \odot (W^{l+1})^T \delta^{l+1}$$

Gradients wrt Parameters



$$\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$ ($w^l_{j,k}$ weight on connection from k^{th} unit in layer l-1 to j^{th} unit in layer l)

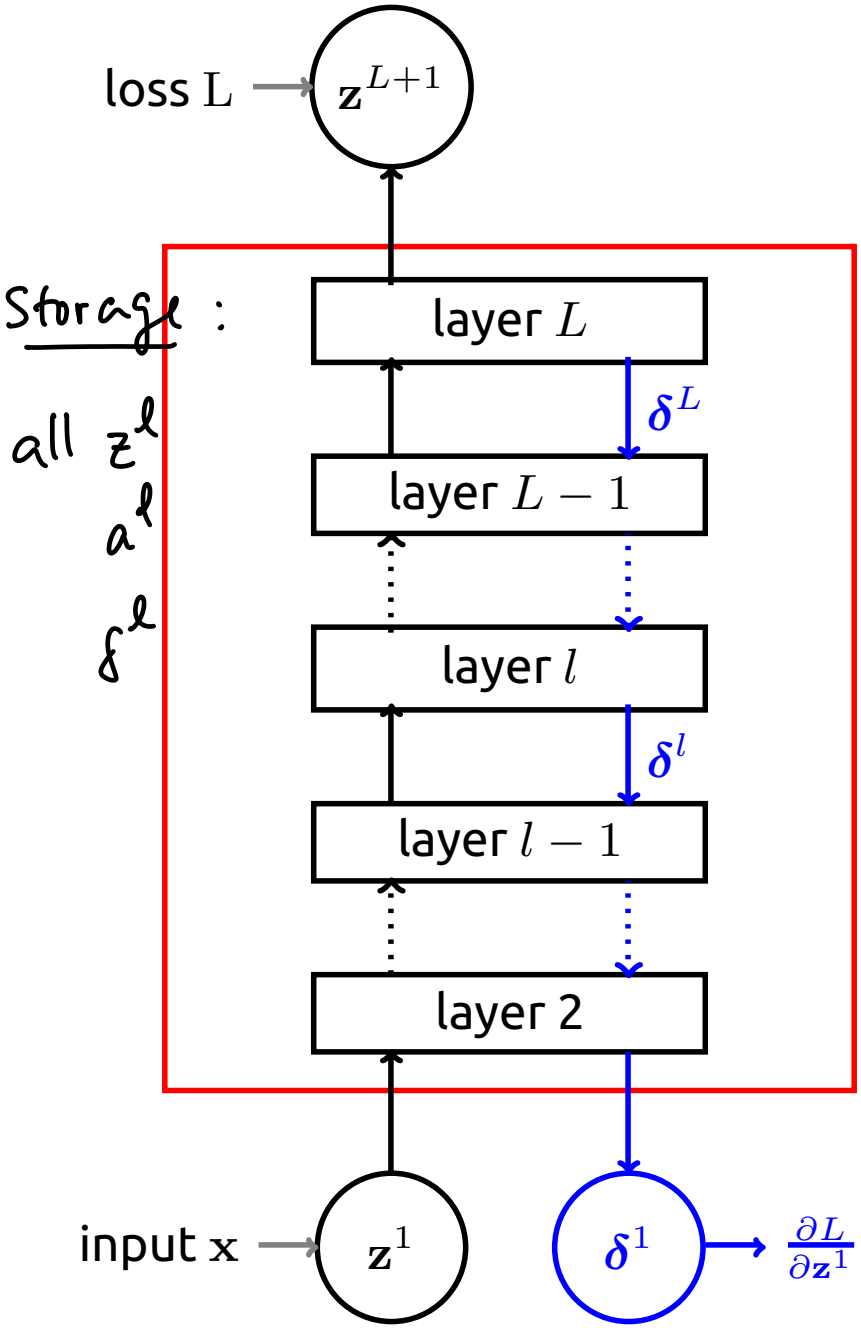
$$\boldsymbol{\delta}^l = \frac{\partial \mathrm{L}}{\partial \mathbf{z}^l}$$

$$\frac{\partial L}{\partial w^l_{ij}} = \frac{\partial L}{\partial z^l_i} \cdot a^{l-1}_j$$

$$= \delta^l_i \cdot a^{l-1}_j$$

$$\frac{\partial L}{\partial W} = \delta^l \cdot (a^{l-1})^T$$

$$z^l_i = w^l_{i1} \cdot a^{l-1}_1 + \cdots + w^l \cdot a^{l-1}_{n_{l-1}} + b^l_i$$



Forward Equations

(1) $\mathbf{z}^1 = \mathbf{x}$ (input)

(2) $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$

(3) $\mathbf{a}^l = f(\mathbf{z}^l)$

(4) $\mathrm{L}(\mathbf{a}^L, y)$

Save on time, but pay a price in space.

Back-propagation Equations

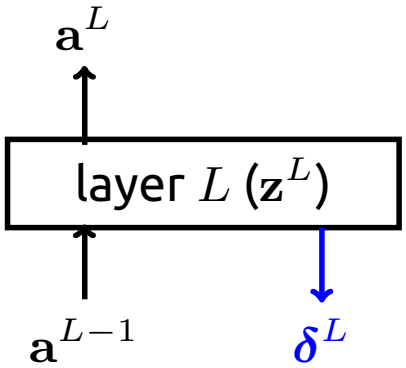
(1) $\boldsymbol{\delta}^L = \frac{\partial \mathrm{L}}{\partial \mathbf{a}^L} \odot f'(\mathbf{z}^L)$

(2) $\boldsymbol{\delta}^l = (\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{l+1} \odot f'(\mathbf{z}^l)$

(3) $\frac{\partial \mathrm{L}}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l$

(4) $\frac{\partial \mathrm{L}}{\partial \mathbf{W}^l} = \boldsymbol{\delta}^l (\mathbf{a}^{l-1})^T$

Output Layer : LogSoftmax




```
criterion = nn.ClassNLLCriterion();
```

Loss: $\mathrm{L}(y, \mathbf{a}^L) = -a^L_y$

$\mathbf{a}^L = \texttt{LogSoftmax}(\mathbf{z}^L)$

$\boldsymbol{\delta}^L = \frac{\partial \mathrm{L}}{\partial \mathbf{z}^L} = \left(\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}\right)^T \cdot \frac{\partial \mathrm{L}}{\partial \mathbf{a}^L}$

$\frac{\partial L}{\partial a^L_y} = -1$; $\frac{\partial L}{\partial a^L_c} = 0$ for $c \neq y$

$a^L_i = z^L_i - \log(\zeta)$; where $\zeta = \sum_j e^{z^L_j}$

$\frac{\partial a^L_i}{\partial z^L_i} = 1 - \frac{e^{z^L_i}}{\zeta} = 1 - p_i$; $p_i = \frac{e^{z^L_i}}{\zeta}$ (prob $\hat{y} = i$)

$\frac{\partial a^L_i}{\partial z^L_j} = -\frac{e^{z^L_j}}{\zeta} = -p_j$

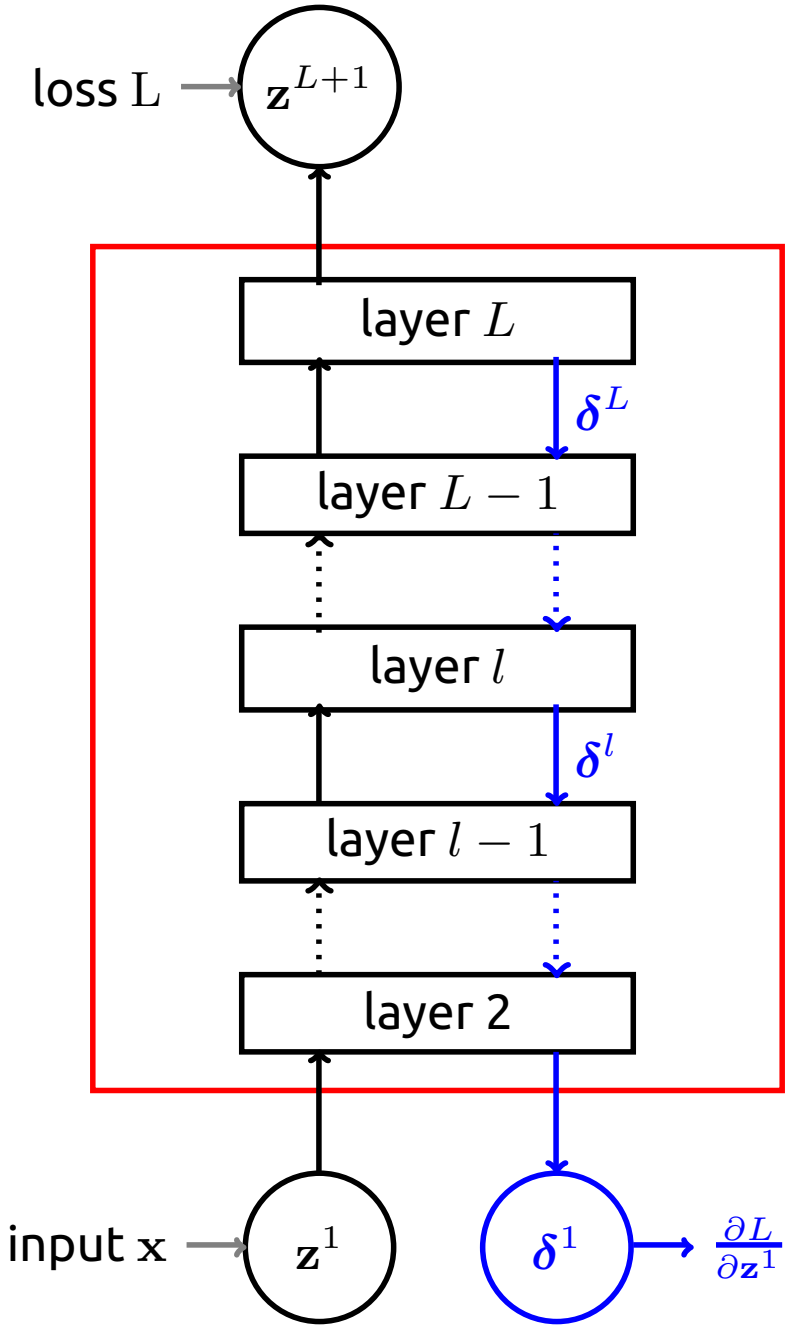
*

$\delta^L_i = p_i \quad \forall i \neq y$

$\delta^L_y = p_i - y$

[Remember multi-class logistic]

*



Forward Equations

(1) $\mathbf{z}^1 = \mathbf{x}$ (input)

(2) $\mathbf{z}^l = g^l(\mathbf{a}^{l-1}; \mathbf{W}^l, \mathbf{b}^l)$

(3) $\mathbf{a}^l = f^l(\mathbf{z}^l)$

(4) $\mathrm{L}(\mathbf{a}^L, y)$

Here g^l is some fn parametrized by W^l, b^l

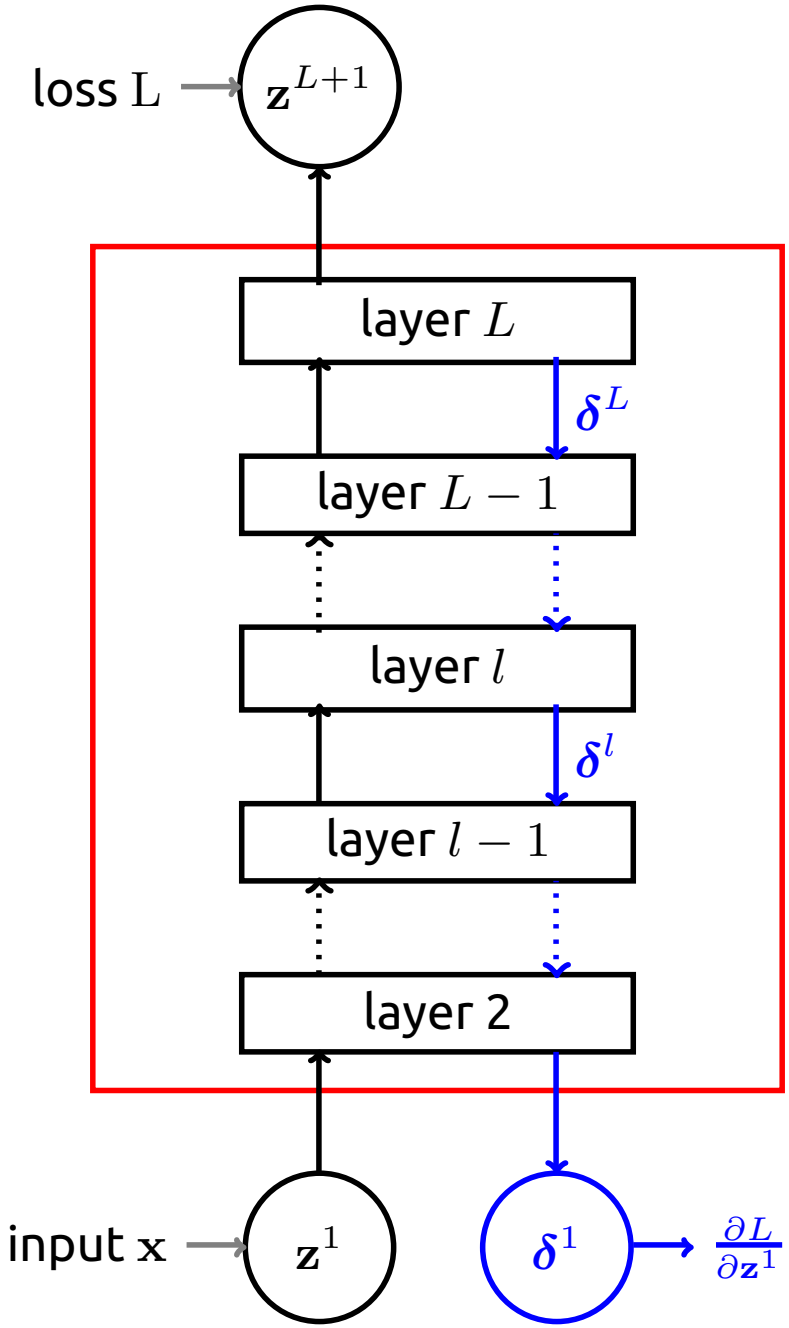
Back-propagation Equations

(1) $\boldsymbol{\delta}^L = \left(\frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L}\right)^T \cdot \frac{\partial \mathrm{L}}{\partial \mathbf{a}^L}$

(2) $\boldsymbol{\delta}^l = \left(\frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{a}^l} \cdot \frac{\partial \mathbf{a}^l}{\partial \mathbf{z}^l}\right)^T \cdot \boldsymbol{\delta}^{l+1}$

(3) $\frac{\partial \mathrm{L}}{\partial \mathbf{b}^l} = \left(\frac{\partial \mathbf{z}^l}{\partial \mathbf{b}^l}\right)^T \cdot \boldsymbol{\delta}^l$

(4) $\frac{\partial \mathrm{L}}{\partial \mathbf{W}^l} = \left(\frac{\partial \mathbf{z}^l}{\partial \mathbf{W}^l}\right)^T \cdot \boldsymbol{\delta}^l$



Forward Equations

(1) $\mathbf{z}^1 = \mathbf{x}$ (input)

(2) $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$

(3) $\mathbf{a}^l = f(\mathbf{z}^l)$

(4) $\mathrm{L}(\mathbf{a}^L, y)$

pre-activation

Linear

non-linear

activation

Back-propagation Equations

(1) $\boldsymbol{\delta}^L = \frac{\partial \mathrm{L}}{\partial \mathbf{a}^L} \odot f'(\mathbf{z}^L)$

(2) $\boldsymbol{\delta}^l = (\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{l+1} \odot f'(\mathbf{z}^l)$ $= \frac{\partial L}{\partial \mathbf{z}^l}$

(3) $\frac{\partial \mathrm{L}}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l$

(4) $\frac{\partial \mathrm{L}}{\partial \mathbf{W}^l} = \boldsymbol{\delta}^l (\mathbf{a}^{l-1})^T$

Computational Questions

What is the running time to compute the gradient for a single data point?

As many matrix multiplications as there are linear layers. (Twice – once in the forward & once in the backward pass)

What is the space requirement?

Vectors z^l, a^l, δ^l

Can we process multiple examples together?

Yes minibatching is useful to take advantage of fast tensor operations, But make sure that all parameters can fit in GPU memory.

```lua
model = nn.Sequential()
model:add(nn.Linear(3, 4))
model:add(nn.Sigmoid())
model:add(nn.Linear(4, 2))
model:add(nn.Sigmoid())
criterion = nn.MSECriterion()
```

Layer 1

Layer 2

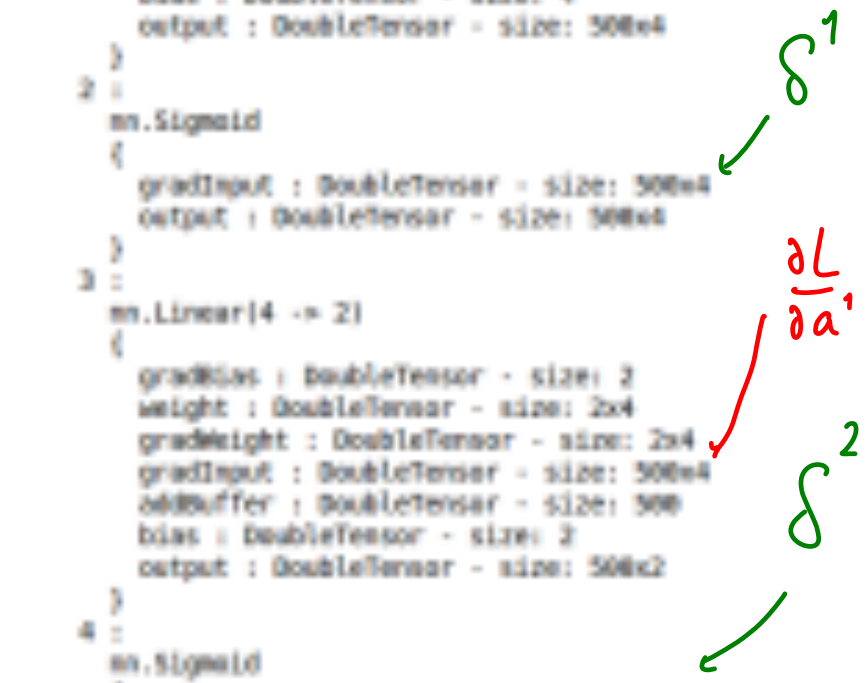
```lua
model.modules
```

```
{
  1 :
    nn.Linear(3 -> 4)
    {
      gradBias : DoubleTensor - size: 4
      weight : DoubleTensor - size: 4x3
      gradWeight : DoubleTensor - size: 4x3
      gradInput : DoubleTensor - empty
      bias : DoubleTensor - size: 4
      output : DoubleTensor - empty
    }
  2 :
    nn.Sigmoid
    {
      gradInput : DoubleTensor - empty
      output : DoubleTensor - empty
    }
  3 :
    nn.Linear(4 -> 2)
    {
      gradBias : DoubleTensor - size: 2
      weight : DoubleTensor - size: 2x4
      gradWeight : DoubleTensor - size: 2x4
      gradInput : DoubleTensor - empty
      bias : DoubleTensor - size: 2
      output : DoubleTensor - empty
    }
  4 :
    nn.Sigmoid
    {
      gradInput : DoubleTensor - empty
      output : DoubleTensor - empty
    }
}
```

```
model = nn.Sequential()
model:add(nn.Linear(3, 4))
model:add(nn.Sigmoid())
model:add(nn.Linear(4, 2))
model:add(nn.Sigmoid())
criterion = nn.MSECriterion()
inputs = torch.randn(500, 3)
targets = torch.randn(500, 2)
model:forward(inputs)
criterion:forward(model.output, targets)
deriv = criterion:backward(model.output, targets)
model:backward(inputs, deriv)
```

```
model.modules
```

```
{
  1 :
    nn.Linear(3 -> 4)
    {
      gradBias : DoubleTensor - size: 4
      weight : DoubleTensor - size: 4x3
      gradWeight : DoubleTensor - size: 4x3
      gradInput : DoubleTensor - size: 500x3
      addBuffer : DoubleTensor - size: 500
      bias : DoubleTensor - size: 4
      output : DoubleTensor - size: 500x4
    }
  2 :
    nn.Sigmoid
    {
      gradInput : DoubleTensor - size: 500x4
      output : DoubleTensor - size: 500x4
    }
  3 :
    nn.Linear(4 -> 2)
    {
      gradBias : DoubleTensor - size: 2
      weight : DoubleTensor - size: 2x4
      gradWeight : DoubleTensor - size: 2x4
      gradInput : DoubleTensor - size: 500x4
      addBuffer : DoubleTensor - size: 500
      bias : DoubleTensor - size: 2
      output : DoubleTensor - size: 500x2
    }
  4 :
    nn.Sigmoid
    {
      gradInput : DoubleTensor - size: 500x2
      output : DoubleTensor - size: 500x2
    }
}
```

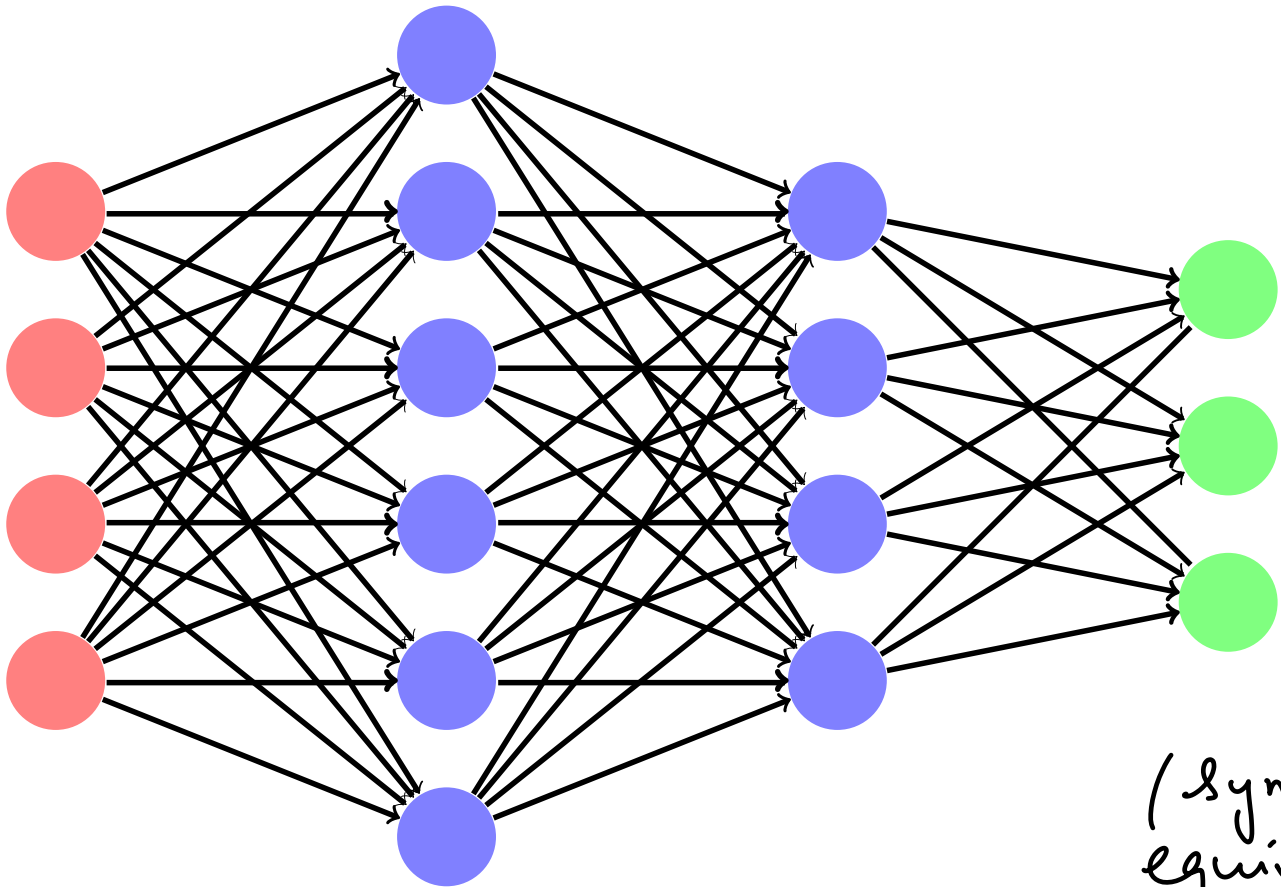


Training Deep Neural Networks

- Back-propagation gives gradient
- Stochastic gradient descent is the method of choice
- Regularisation
  - How do we add ℓ_1 or ℓ_2 regularisation?
  - Don't regularise bias terms
- How about convergence?
- What did we learn in the last 10 years, that we didn't know in the 80s?

Training Feedforward Deep Networks

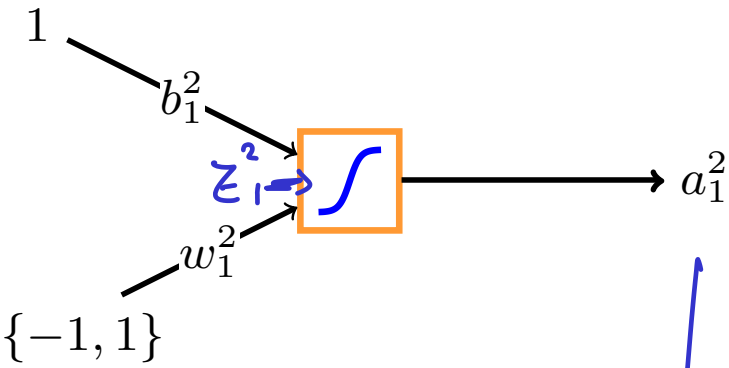
Layer 1 (Input) | Layer 2 (Hidden) | Layer 3 (Hidden) | Layer 4 (Output)



(Symmetries lead to equivalent weights (eg any permutation of hidden units.) We are more worried about BAD local minima.

Why do we get non-convex optimisation problem?

A toy example



Target is $y = \frac{1-x}{2}$

$x = +1 \quad y = 0$

$x = -1 \quad y = 1$

$z_1^2 = w_1^2 x + b_1^2$

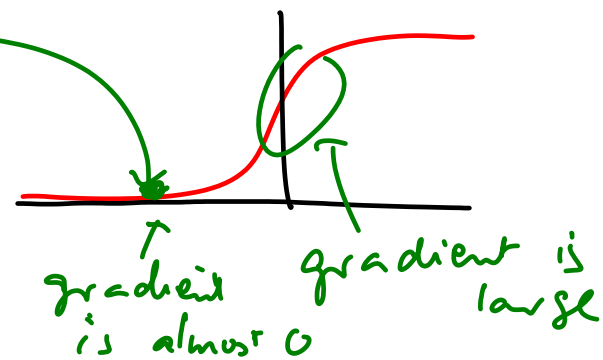
$L = \frac{1}{2}(y - \sigma(z_1^2))^2$

$\frac{\partial L}{\partial z_1^2} = (y - \sigma(z_1^2))\,\sigma'(z_1^2)$

$x = -1$

$w_1^2 = 2, \; b_1^2 = 0.5$

$z_1^2 = -2.5$



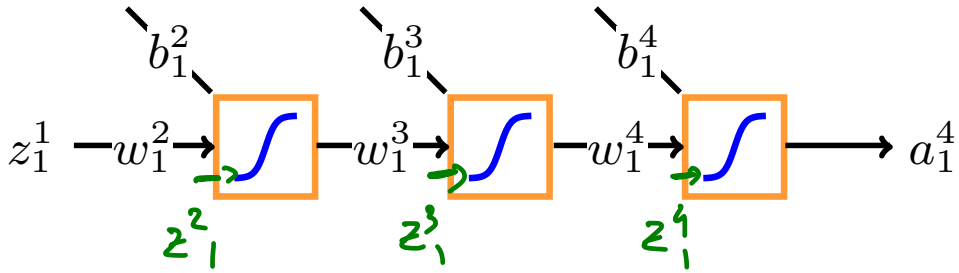
CROSS ENTROPY

$L = -(y \log(\sigma(z_1^2)) + (1-y)\log(1-\sigma(z_1^2)))$

Say $x = -1, \; y = 1,$

then $\frac{\partial L}{\partial z_1^2} = 1 - \sigma(z_1^2)$

Propagating Gradients Backwards



(SATURATION)

$z_1^4 = w_1^4 \sigma(z_1^3) + b_1^4$; L : CROSS ENTROPY.

$\frac{\partial L}{\partial z_1^4} = y - \sigma(z_1^4)$

$\frac{\partial L}{\partial z_1^3} = \frac{\partial L}{\partial z_1^4} \cdot \left(w_1^4 \cdot \sigma'(z_1^3) \right)$

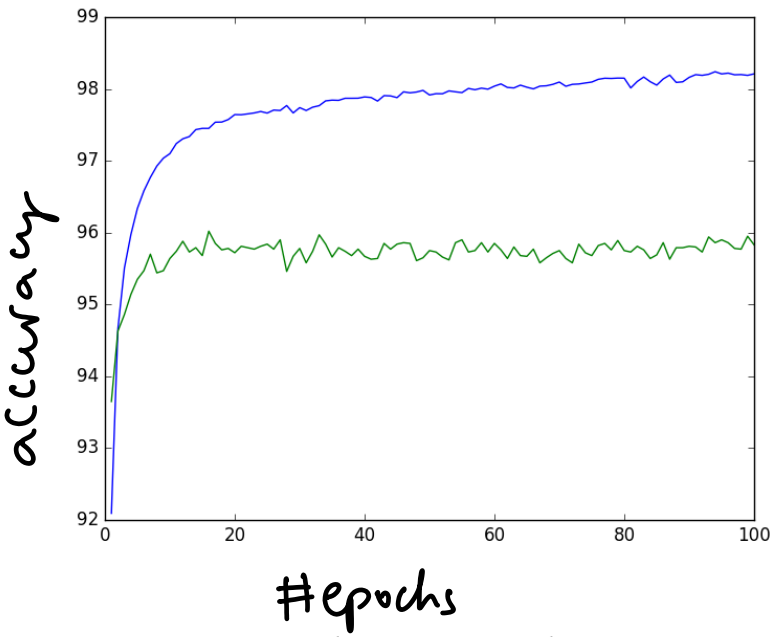
$\frac{\partial L}{\partial z_1^2} = \frac{\partial L}{\partial z_1^4} \cdot (w_1^4 \cdot \sigma'(z_1^3))(w_1^3 \cdot \sigma'(z_1^2))$

(VANISHING GRADIENT)

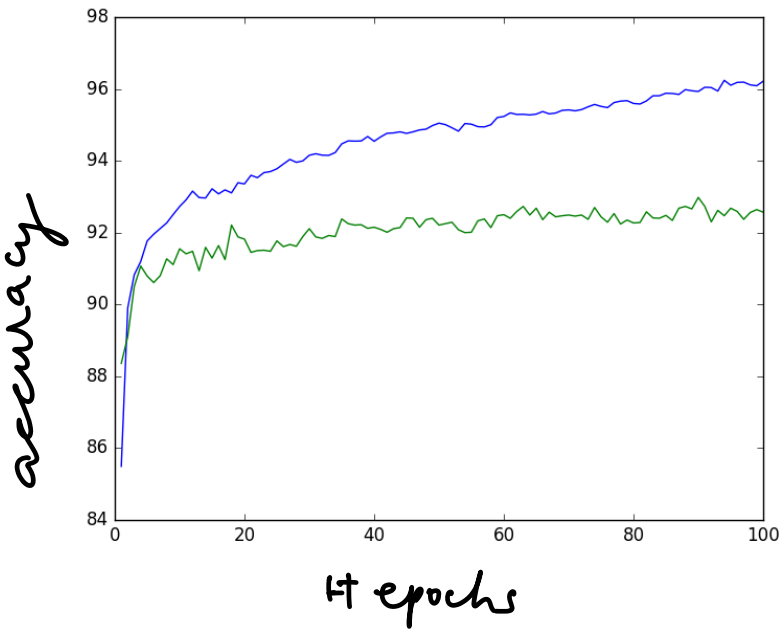
$\sigma' \in (0, 1/4)$

← gradient across many layers can go to 0

Digit Classification Problem on Sheet 4



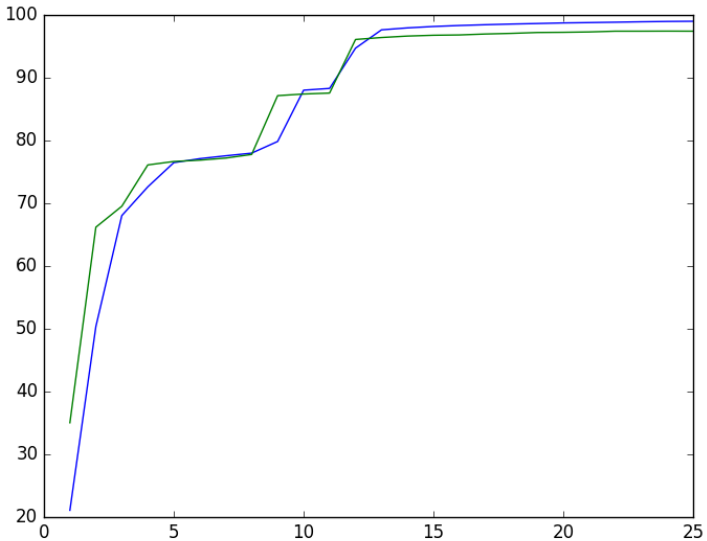
One-hot encoding



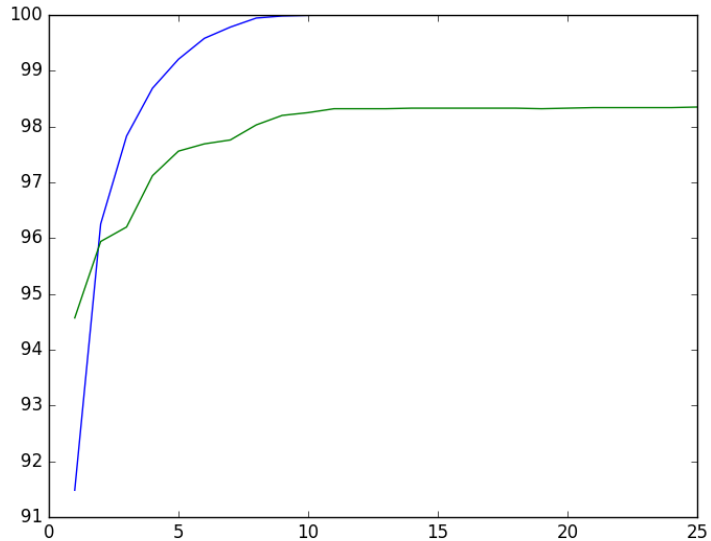
Binary encoding

Digit Classification: Squared Loss vs Cross Entropy

uses 1024 hidden units



Squared Error



Cross Entropy

(See paper by Glorot and Bengio (2010))

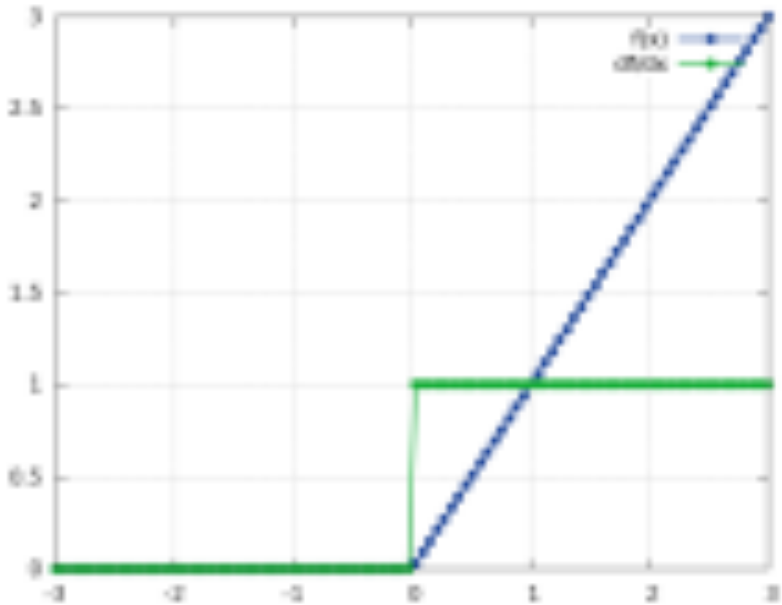
Avoiding Saturation

Use *rectified linear units*

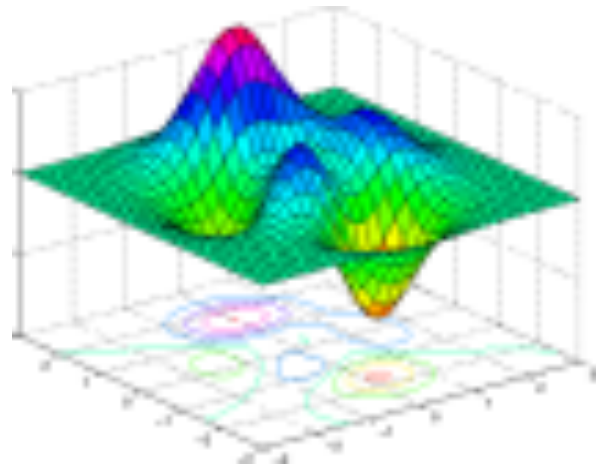
$\mathrm{ReLU}(z) = \max(0, z)$

Sometimes, you will see just $f(z) = |z|$

Other varieties: leaky ReLUs, parametric ReLUs



Initialising Weights and Biases



Why is initialising important?

Non-convexity. Initialization can affect which local minima is reached.

Suppose we were using a *sigmoid unit*, how would you initialise the incoming weights?

Small values. If $z = \sum_{i=1}^{n} w_i a_i$

What if it were a ReLU unit? e.g., choose $w_i \in \left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$ at random.

How about the biases?

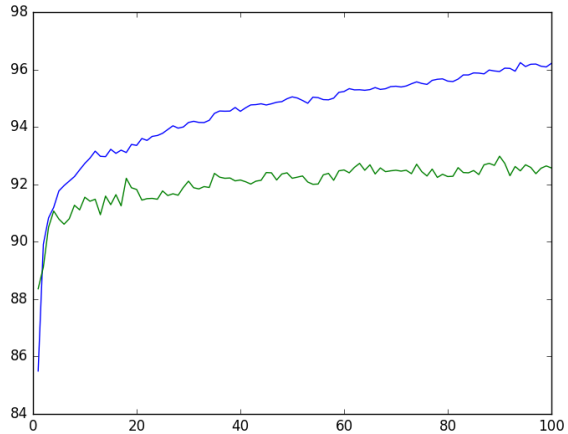
Avoiding Overfitting

Deep Neural Networks have a lot of parameters

- Fully connected layers with $n_1, n_2, .., n_k$ units have at least $n_1 n_2 + n_2 n_3 + \cdots + n_{k-1} n_k$ parameters
- MLP for digit recognition had 2 million parameters!
- For image detection, the neural net used by Krizhevsky, Sutskever, Hinton (2012) has 60 *million* parameters and 1.2 *million* training images
- How do we avoid deep neural networks from overfitting?

Early Stopping

Maintain validation set and stop training when error on validation set stops decreasing.



What are the computational costs?

(i) maintain and compute validation error

(ii) store model when necessary

What are the advantages?

(i) efficient, compared to other methods

Early stopping leads to better generalisation

(See paper by Hardt, Recht and Singer (2015))

Add Data

Typically, getting additional data is either impossible or expensive

Fake the data!

Images can be translated slight, rotated slightly, change of brightness, etc.

Google Offline Translate trained on entirely fake data!

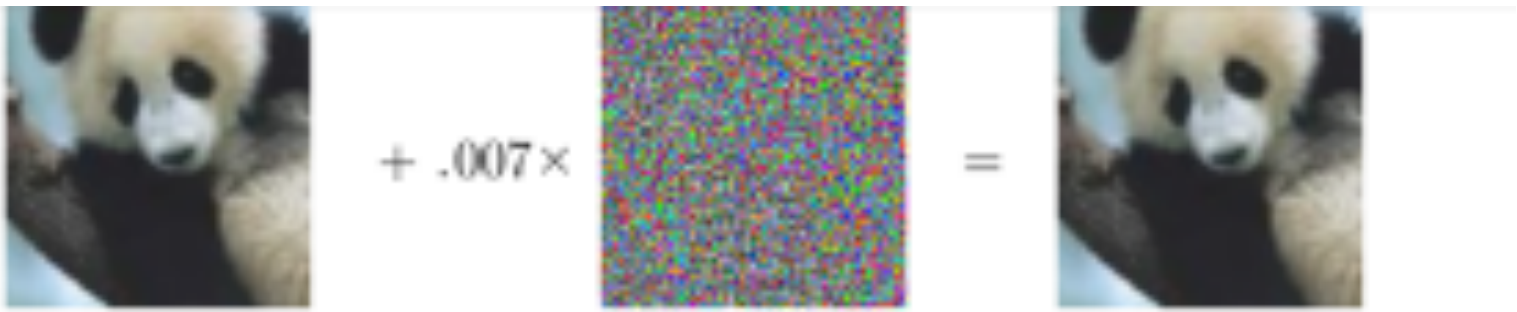
(Google Research Blog)

Add Data

Adversarial Training

Take trained (or partially trained model)

Create examples by modifications “imperceptible to the human eye”, but where the model fails



(Szegedy *et al.*, Goodfellow *et al.*)

Other Ideas to Reduce Overfitting

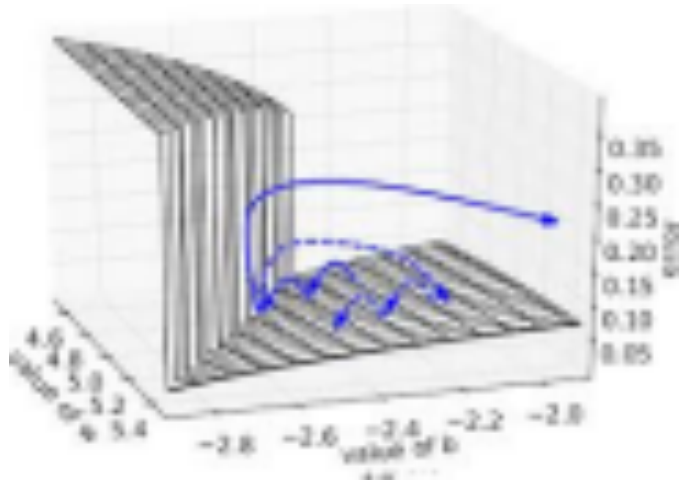
Hard constraints on weights

Gradient Clipping

Inject noise into the system

Enforce sparsity in the neural network

Unsupervised Pre-training



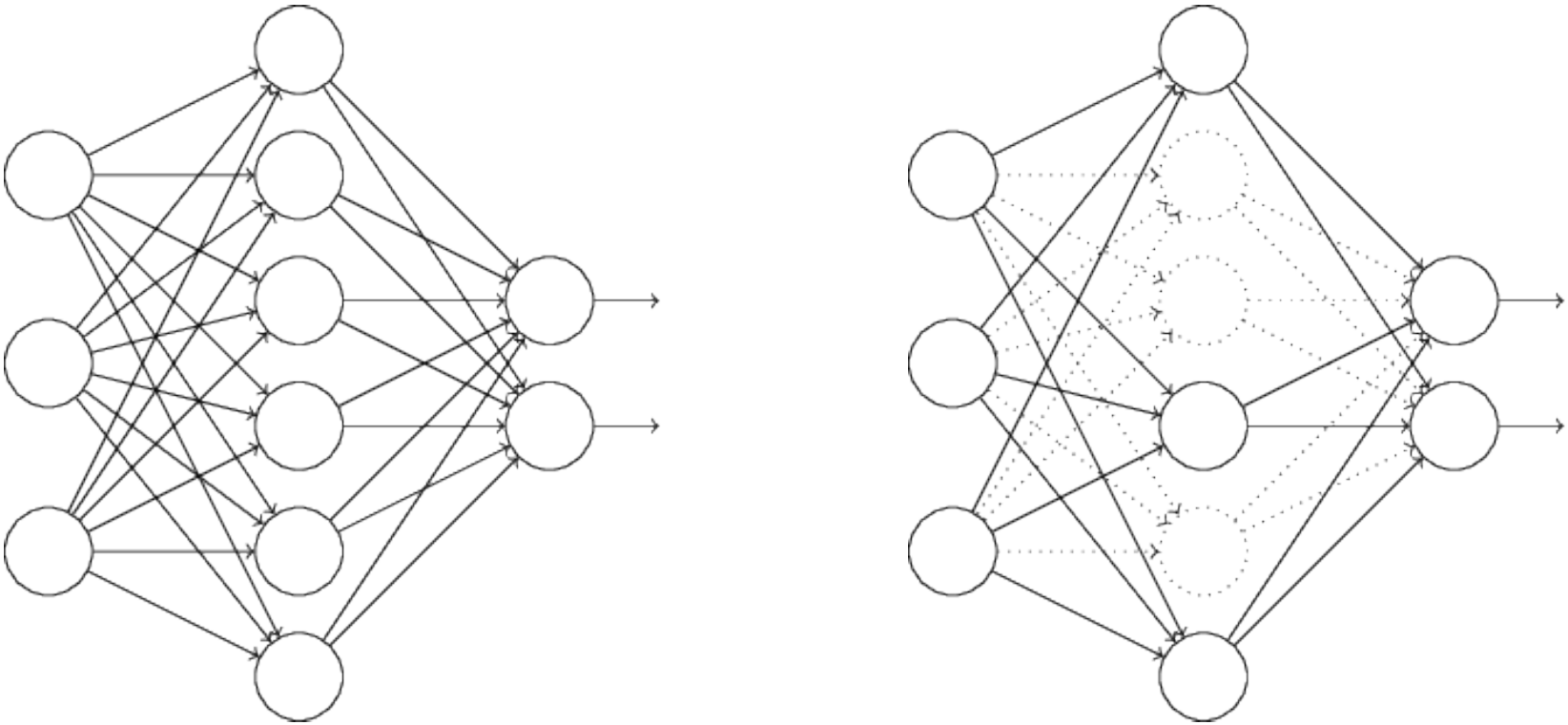
(Bengio *et al.*)

Bagging and Dropout

Bagging (Leo Breiman - 1994)

- Given dataset $\mathcal{D} = \langle(\mathbf{x}_i, y_i)\rangle_{i=1}^m$, sample $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_k$ of size m from $\mathcal{D}$ with replacement
- Train classifiers $f_1, \ldots, f_k$ on $\mathcal{D}_1, \ldots, \mathcal{D}_k$
- When predicting use majority (or average if using regression)
- Clearly this approach is not practical for deep networks

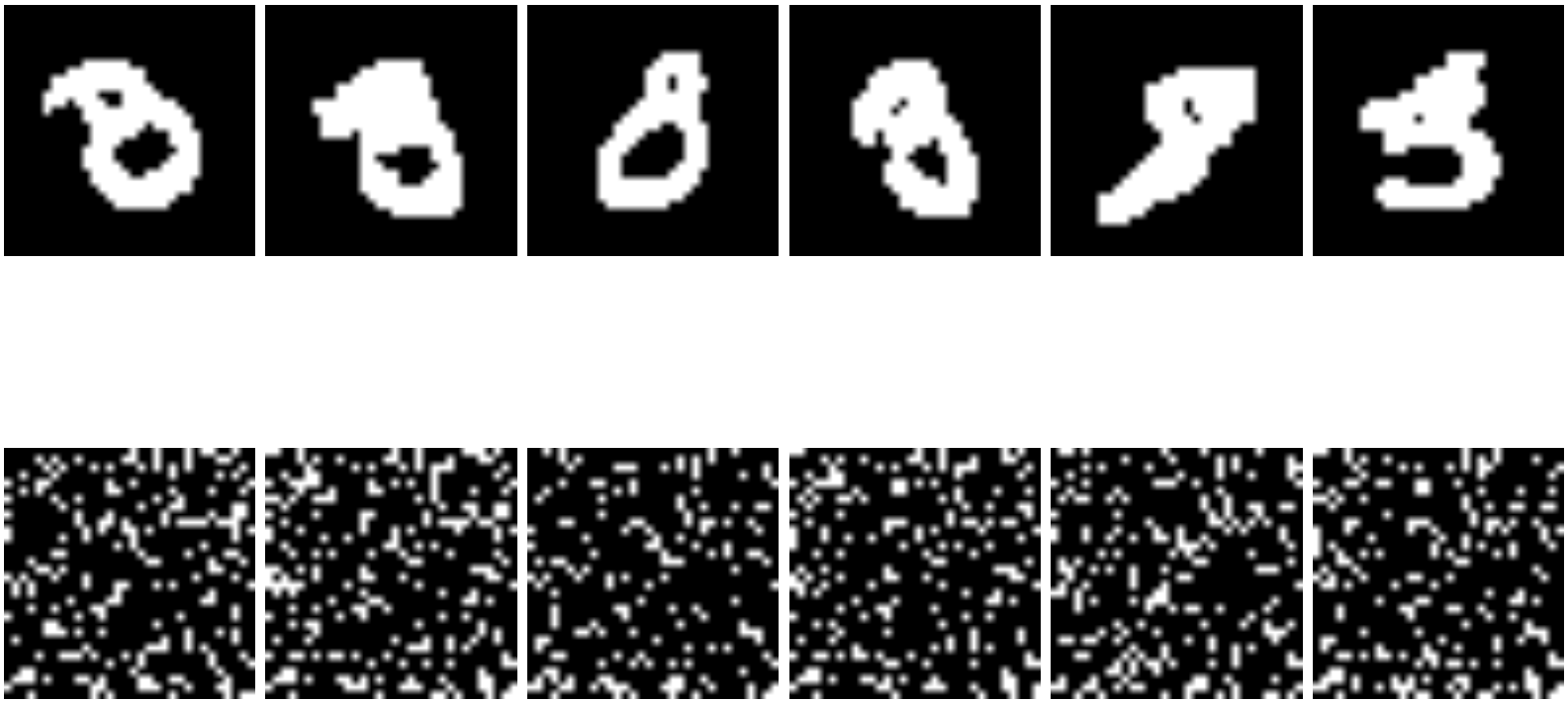
Dropout



- For input x each hidden unit with probability $1/2$ independently
- Every input, will have a potentially different mask
- Potentially exponentially different models, but have "same weights"
- After training whole network is used by halving all the weights

(Srivastava, Hinton, Krizhevsky, 2014)

Errors Made by MLP for Digit Recognition

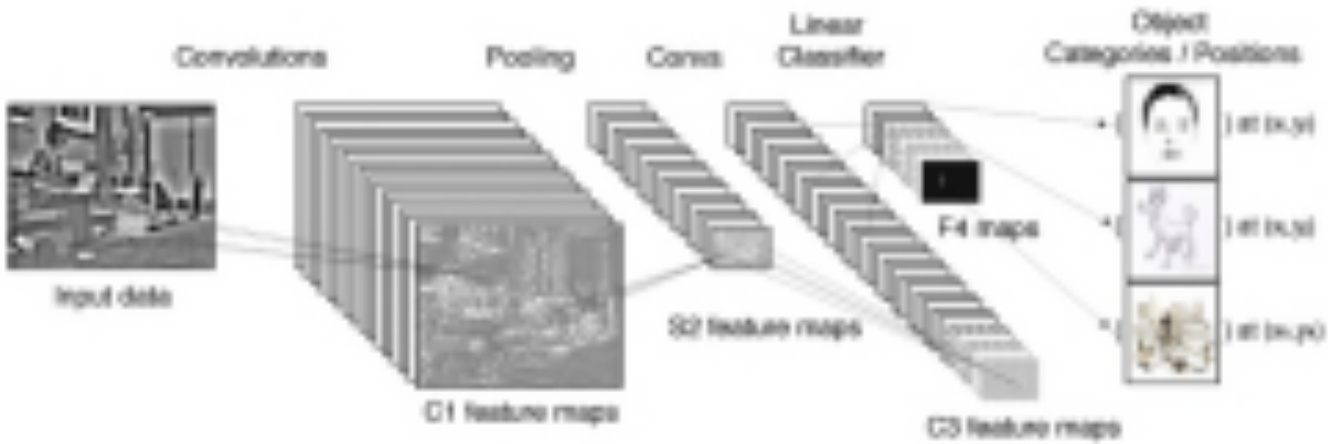


Avoiding Overfitting

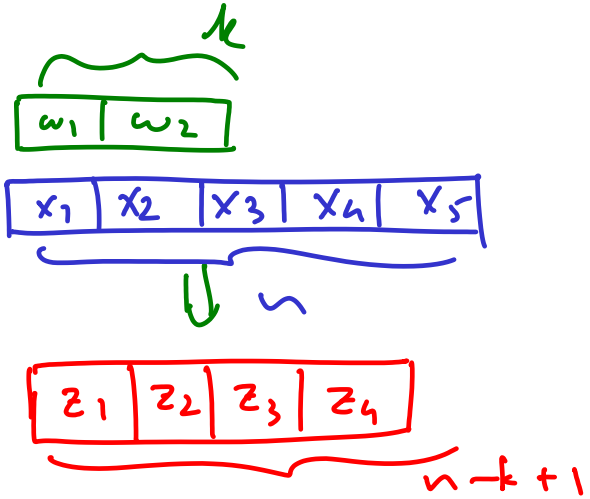
- Use weight sharing a.k.a tied weights in the model
- Exploit invariances -- translation
- Exploit locality in images, audio, text, *etc.*
- Convolutional Neural Networks (convnets)

Convolutional Neural Networks (convnets)

(Fukushima, LeCun, Hinton 1980s)



Convolution



$z_1 = x_1 w_1 + x_2 w_2$

$z_2 = x_2 w_1 + x_3 w_2$

$z_3 = x_3 w_1 + x_4 w_2$

$z_4 = x_4 w_1 + x_5 w_2$

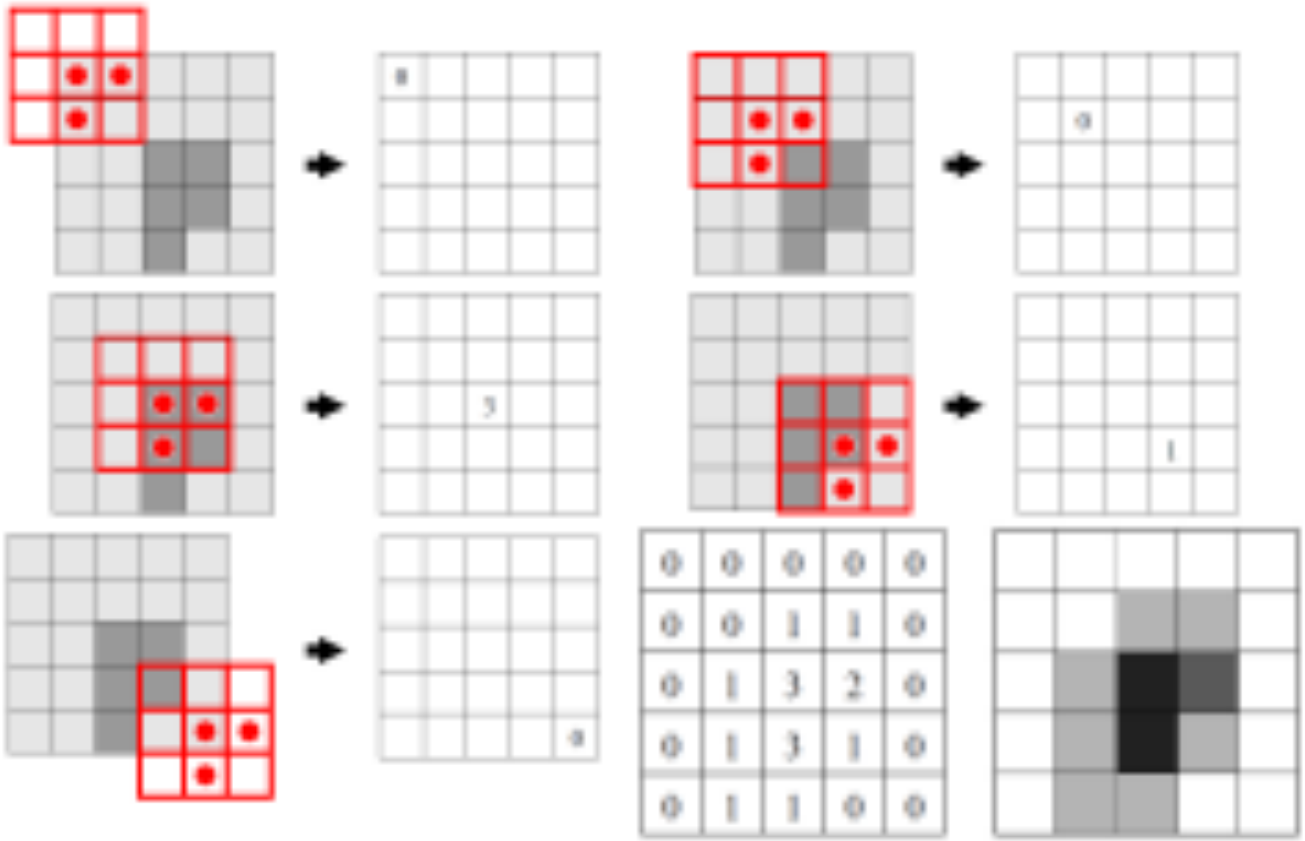
Zero padding to get suitable length.

stride: don't take every consecutive position.

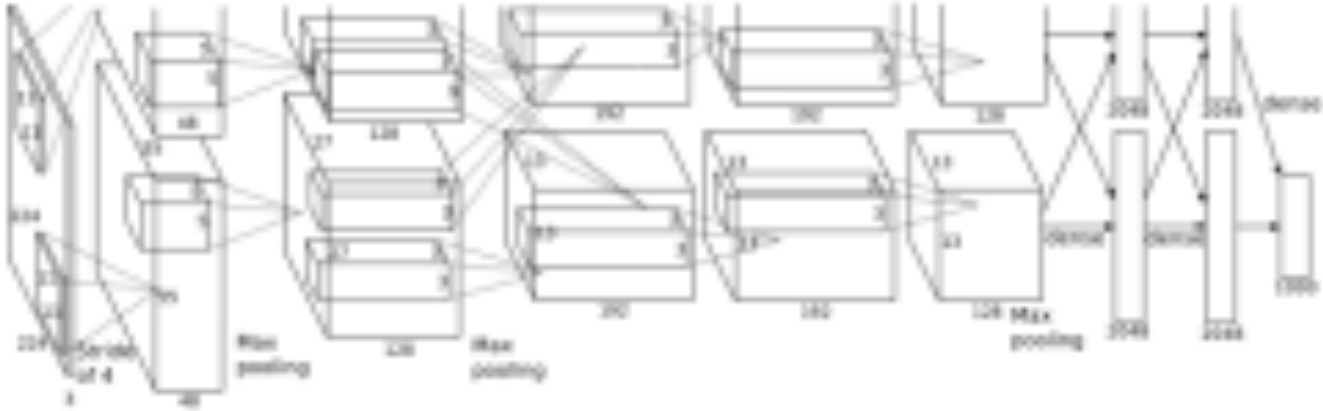
[Mathematically: convolution is defined somewhat differently. This is actually correlation.]

[See: kernel flipping]

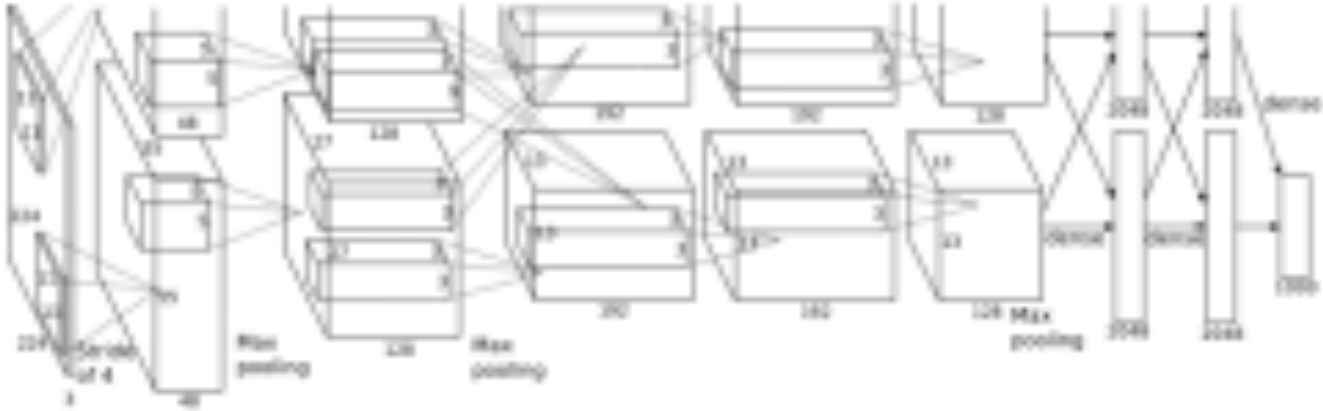
Image Convolution

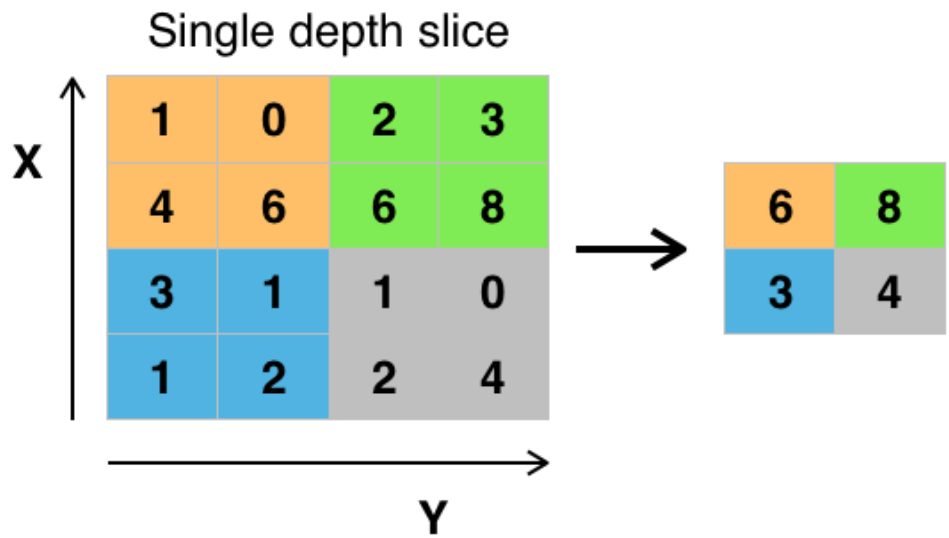


Source: L. W. Kheng

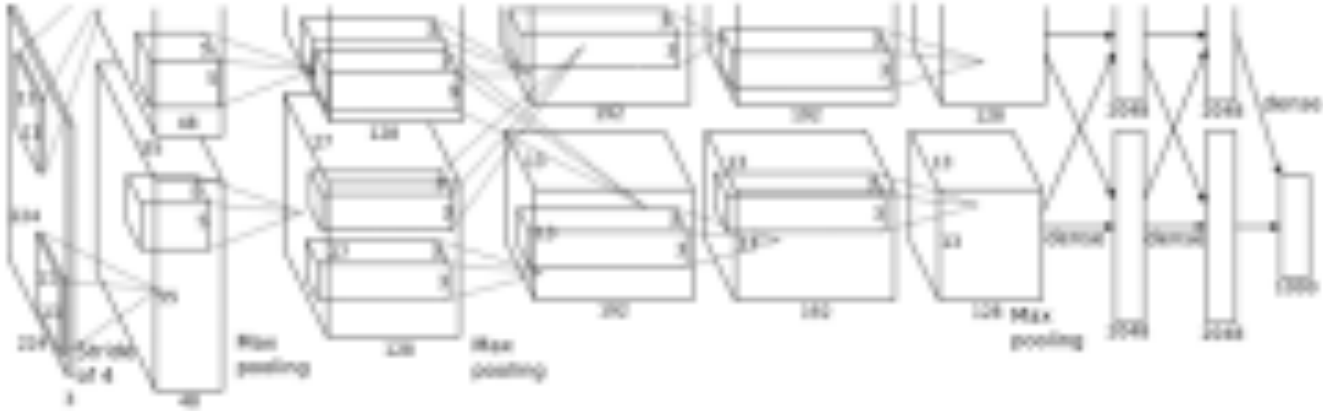


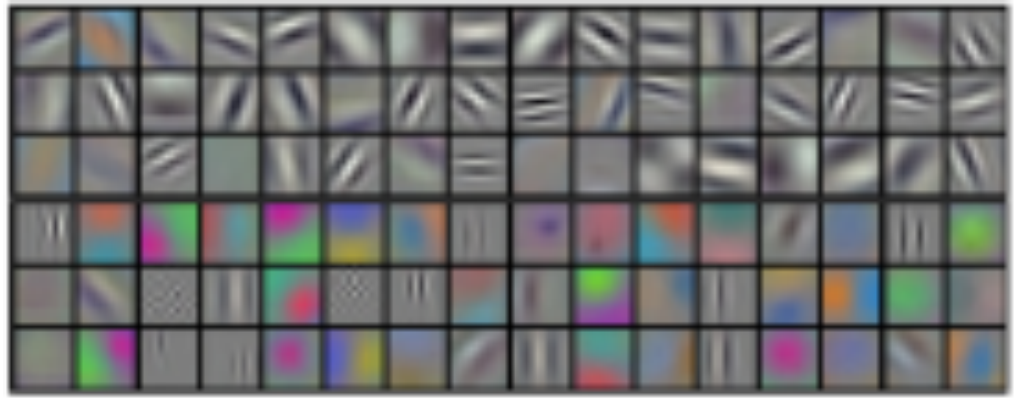
Source: Krizhevsky, Sutskever, Hinton (2012)



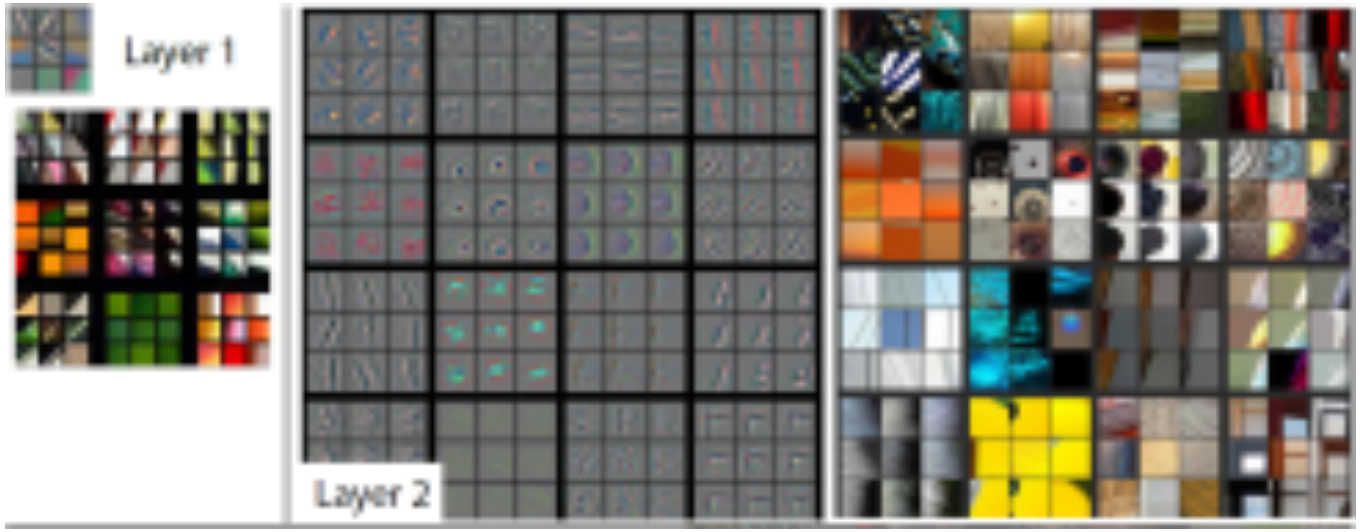


Source: Krizhevsky, Sutskever, Hinton (2012); Wikipedia

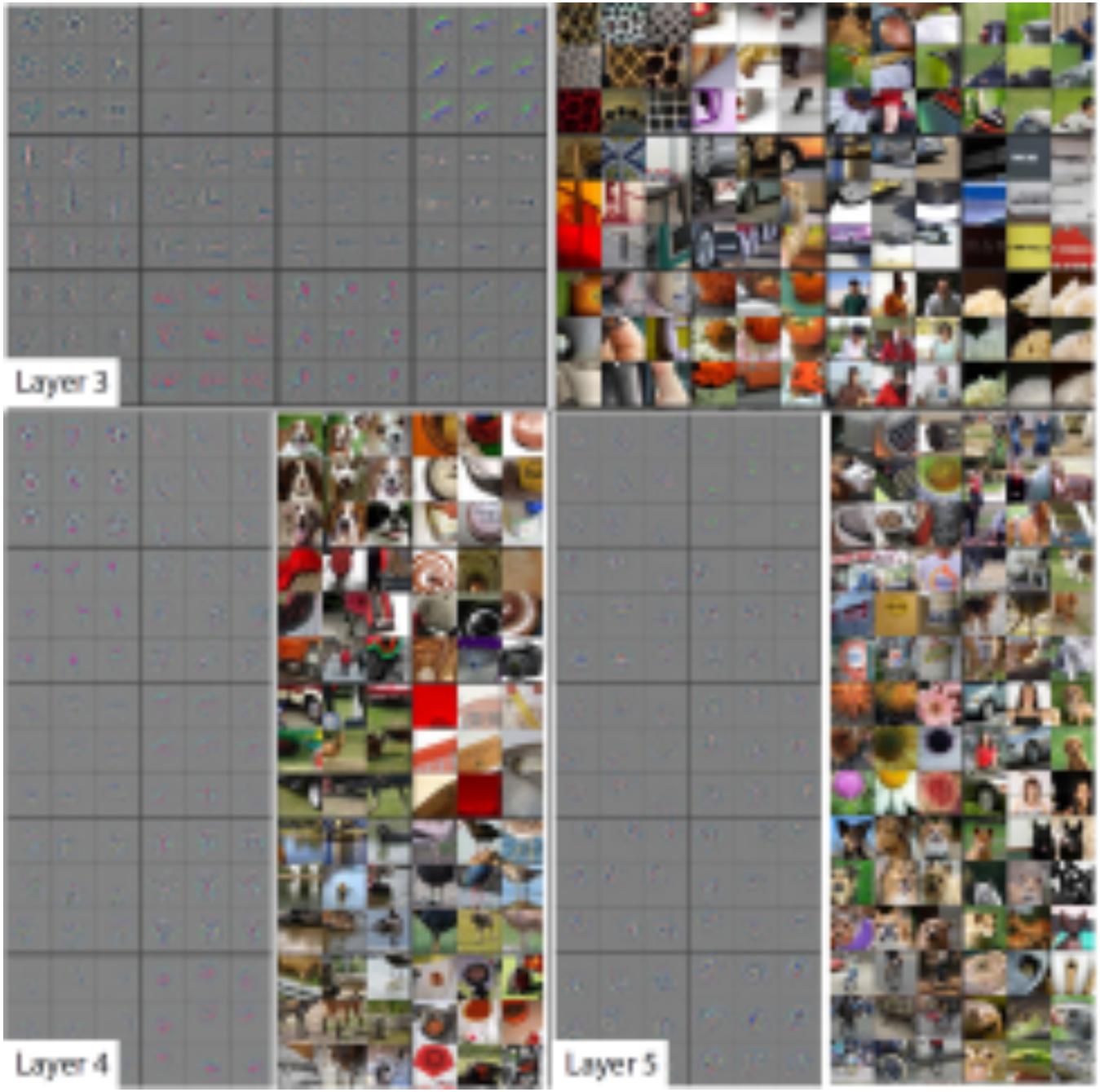




Source: Krizhevsky, Sutskever, Hinton (2012)



Source: Zeiler and Fergus (2013)



Source: Zeiler and Fergus (2013)

Convnet in Torch

convnet

```
model = nn.Sequential()
model:add(nn.Reshape(1, 32, 32))
-- layer 1:
model:add(nn.SpatialConvolution(1, 16, 5, 5))
model:add(nn.ReLU())
model:add(nn.SpatialMaxPooling(2, 2, 2, 2))
-- layer 2:
model:add(nn.SpatialConvolution(16, 128, 5, 5))
model:add(nn.ReLU())
model:add(nn.SpatialMaxPooling(2, 2, 2, 2))
-- layer 3:
model:add(nn.Reshape(128*5*5))
model:add(nn.Linear(128*5*5, 200))
model:add(nn.ReLU())
-- output
model:add(nn.Linear(200, 10))
model:add(nn.LogSoftMax())
```

Convolutional Layer

$$z^{l+1}_{i',j',f'} = b_{f'} + \sum_{i=1}^{W_{f'}} \sum_{j=1}^{H_{f'}} \sum_{f=1}^{F_l} a^{l}_{i'+i-1,j'+j-1,f} w^{l+1,f'}_{i,j,f}$$

$$\frac{\partial z^{l+1}_{i',j',f'}}{\partial w^{l+1,f'}_{i,j,f}} = a^{l}_{i'+i-1,j'+j-1,f}$$

$$\frac{\partial L}{\partial w^{l+1,f'}_{i,j,f}} = \sum_{i',j'} \delta^{l+1}_{i',j',f'} a^{l}_{i'+i-1,j'+j-1,f}$$

Convolutional Layer

$$z^{l+1}_{i',j',f'} = b_{f'} + \sum_{i=1}^{W_{f'}} \sum_{j=1}^{H_{f'}} \sum_{f=1}^{F_l} a^{l}_{i'+i-1,j'+j-1,f} w^{l+1,f'}_{i,j,f}$$

$$\frac{\partial z^{l+1}_{i',j',f'}}{\partial a^{l}_{i,j,f}} = w^{l+1,f'}_{i-i'+1,j-j'+1,f}$$

$$\frac{\partial L}{\partial a^{l}_{i,j,f}} = \sum_{i',j',f'} \delta^{l+1}_{i',j',f'} w^{l+1,f'}_{i-i'+1,j-j'+1,f}$$

Max-Pooling Layer

$$b^{l+1}_{i',j'} = \max_{i,j \in \Omega(i',j')} a^l_{i,j}$$

$$\frac{\partial b^{l+1}_{i',j'}}{\partial a^l_{i,j}} = \mathbb{I}\left((i,j) = \operatorname*{argmax}_{\tilde{i},\tilde{j} \in \Omega(i',j')} a^l_{\tilde{i},\tilde{j}}\right)$$

domain over which pooling is performed.

{The derivative can be undefined when argmax is actually a set, and not a single point}