Deep learning with Neural Networks

- Dense Neural Networks
- Backpropagation
- Regularization
- Normalization layers
- Learning rates
- Feature explainability

Alejandro Lancho Serrano, alancho@ing.uc3m.es

Material original por Pablo Martínez Olmos

Fundaments of Neural Networks

uc3m

What is Deep Learning?

Artificial Intelligence

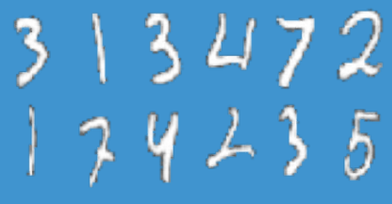
Any technique that enables computers to mimic human behavior

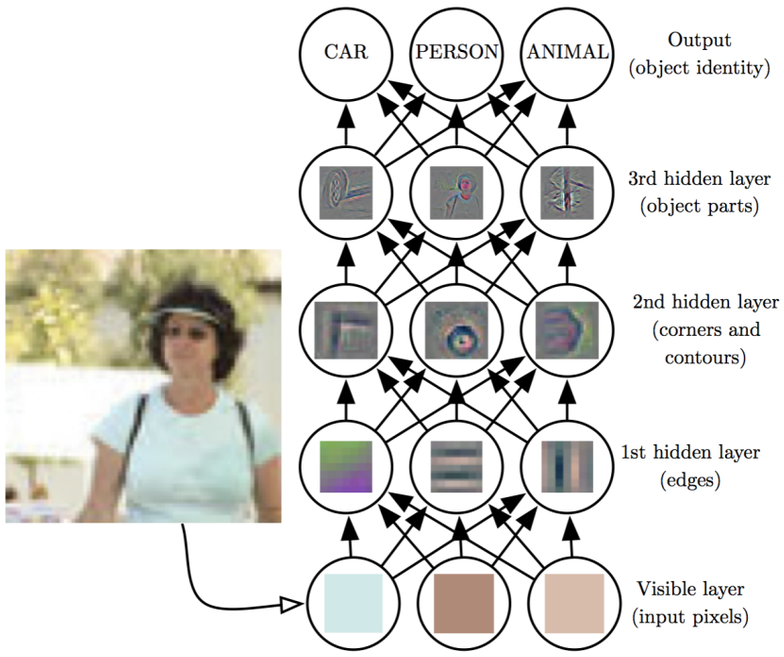
Machine Learning

Ability to learn without explicitly being programmed

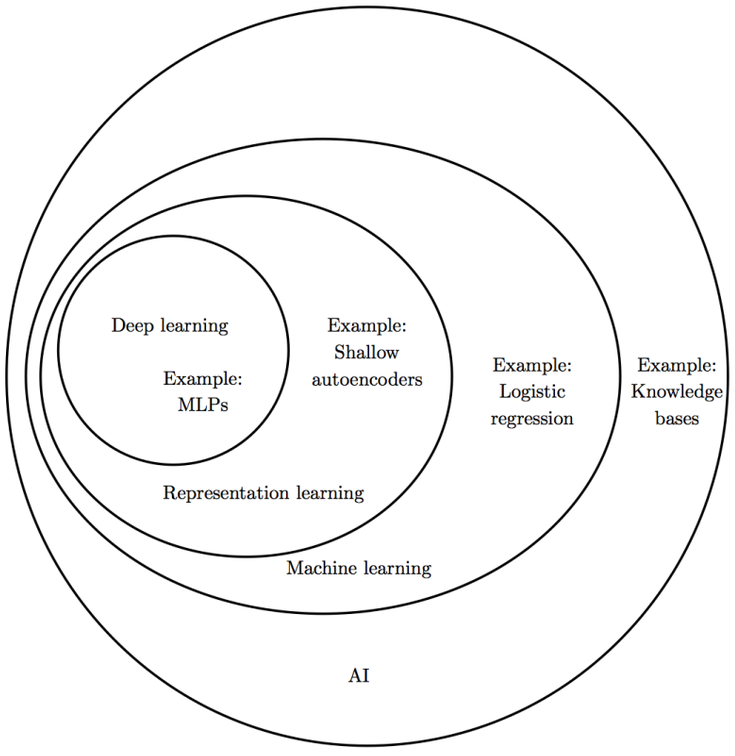
Deep Learning

Extract patterns from data using neural networks





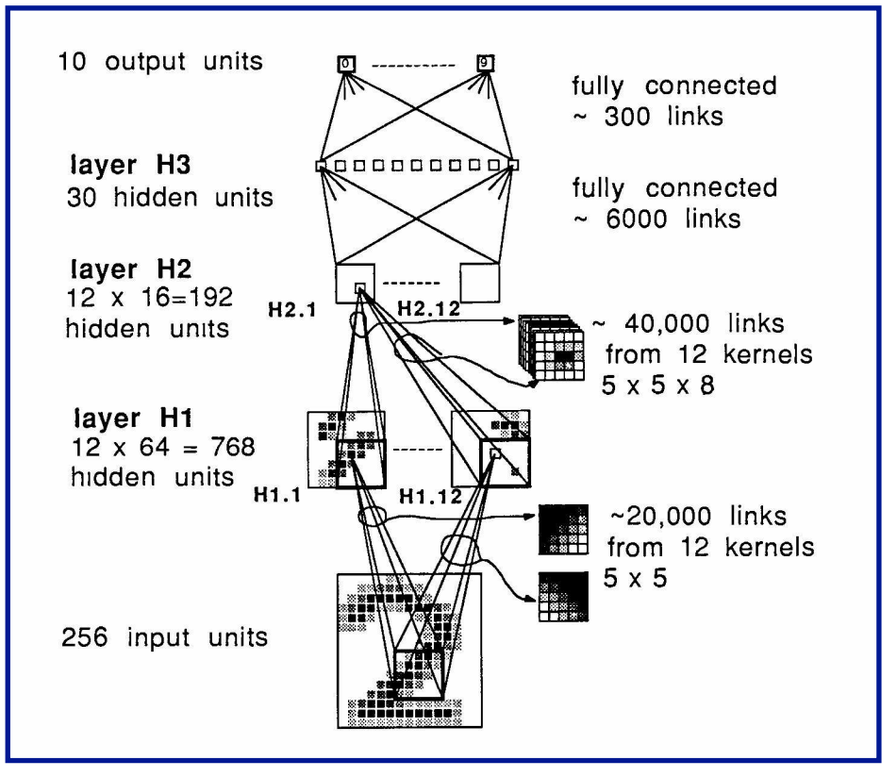
Source: Deep Learning, Good Fellow et al. 2017



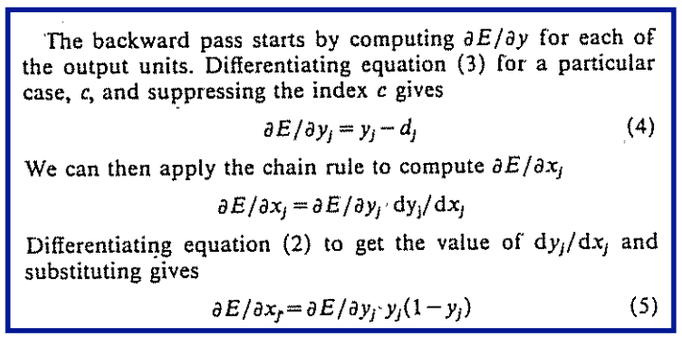
Neural Networks Revisited

Most elements of network architecture employed as early as the mid-1980's

A History of Deep Learning (link)



Backpropagation applied to Handwritten Zip Code Recognition
LeCun et al. (1989)



The backward pass starts by computing $\partial E/\partial y$ for each of the output units. Differentiating equation (3) for a particular case, c, and suppressing the index c gives

$$\partial E/\partial y_j = y_j - d_j \quad (4)$$

We can then apply the chain rule to compute $\partial E/\partial x_j$

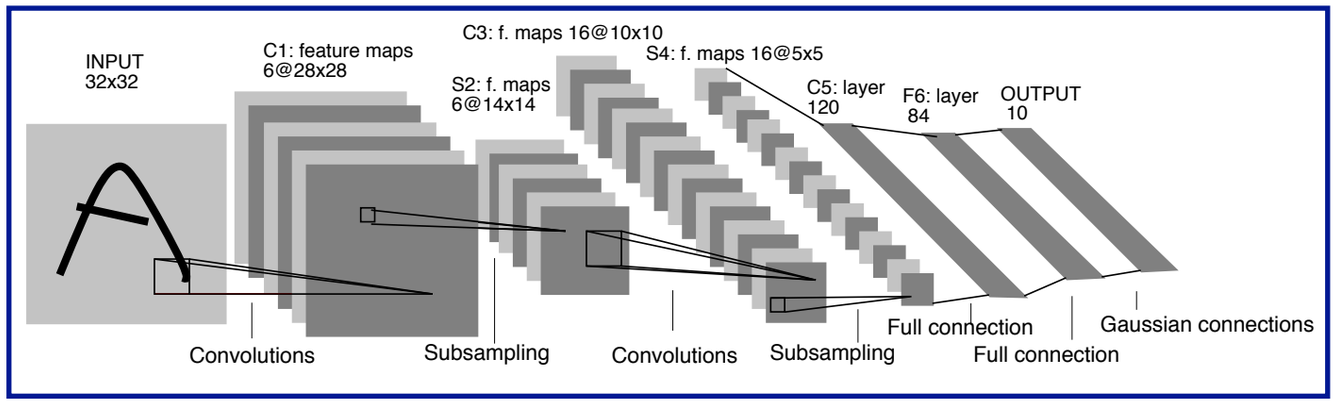
$$\partial E/\partial x_j = \partial E/\partial y_j \cdot \mathrm{d}y_j/\mathrm{d}x_j$$

Differentiating equation (2) to get the value of $\mathrm{d}y_j/\mathrm{d}x_j$ and substituting gives

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j(1-y_j) \quad (5)$$

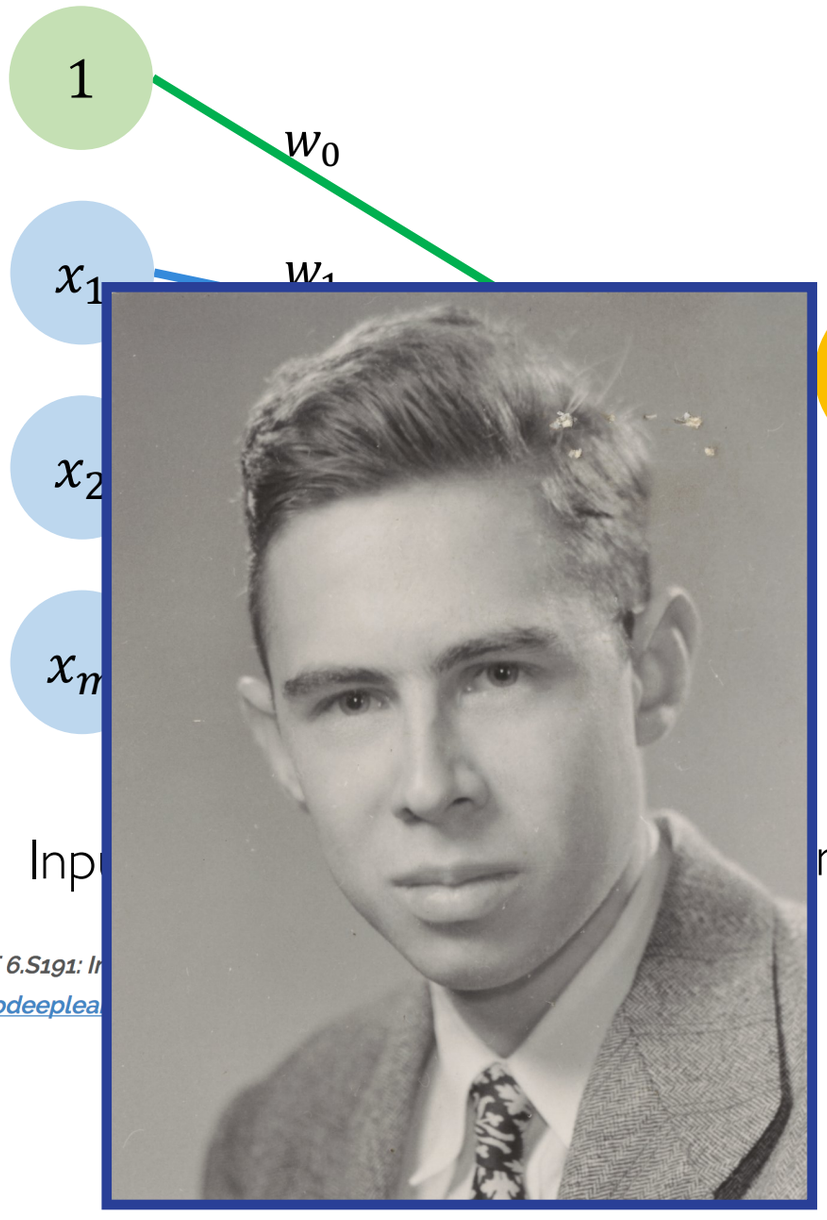
Learning representations
by back-propagating errors

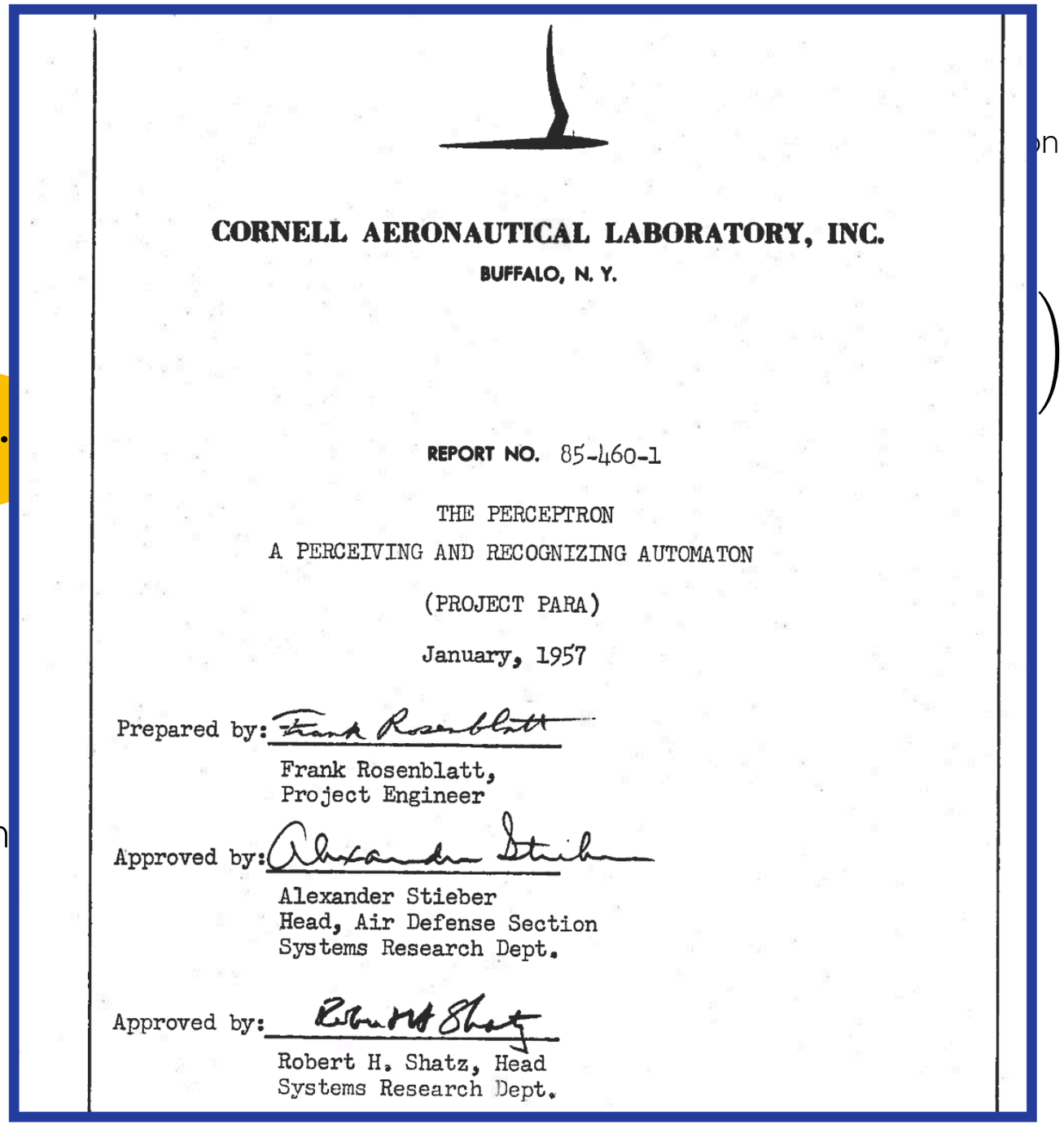
Rumelhart, Hinton, Williams (1986)



Gradient-Based Learning applied to Document Recognition
LeCun, Bengio, Haffner (1998)

The Perceptron: The structural building block of deep learning





CORNELL AERONAUTICAL LABORATORY, INC.
BUFFALO, N. Y.

REPORT NO. 85-460-1

THE PERCEPTRON
A PERCEIVING AND RECOGNIZING AUTOMATON

(PROJECT PARA)

January, 1957

Prepared by: Frank Rosenblatt
Frank Rosenblatt,
Project Engineer

Approved by: Alexander Stieber
Alexander Stieber
Head, Air Defense Section
Systems Research Dept.

Approved by: Robert H. Shatz
Robert H. Shatz, Head
Systems Research Dept.

Inpu n-Lin

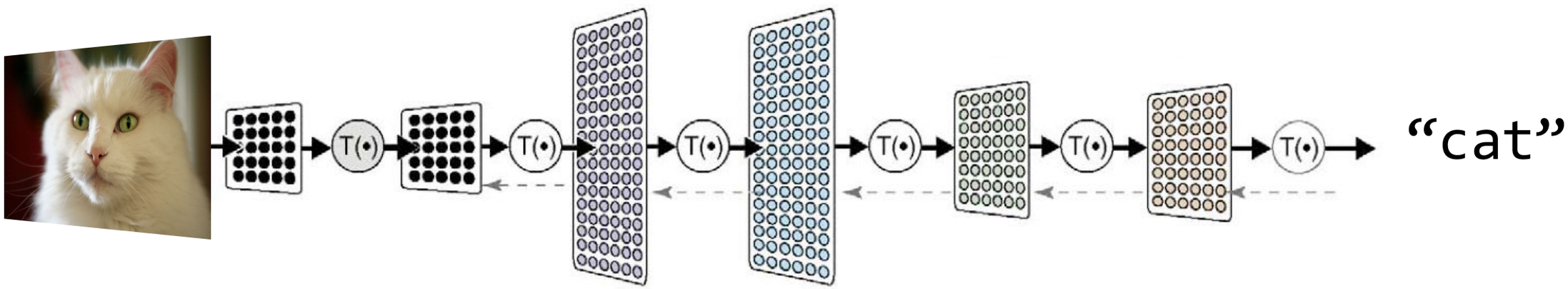
Wikipedia Entry for Frank R.

Dense Neural Networks

uc3m

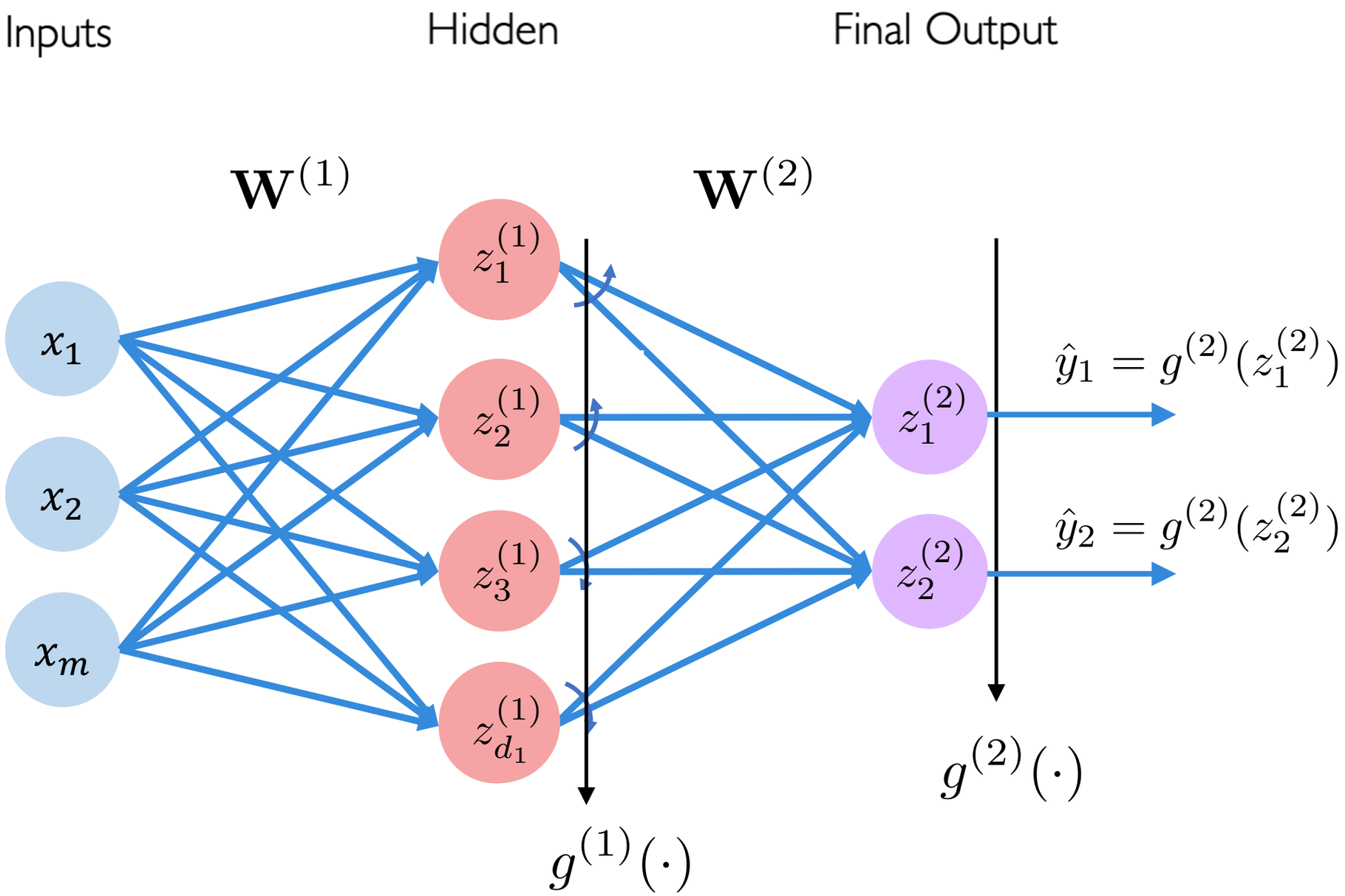
Deep learning = artificial neural networks

Hierarchical composition of **simple mathematical** functions



Untangling invariant object recognition
J DiCarlo and D Cox (2007)

A Neural Network with two layers

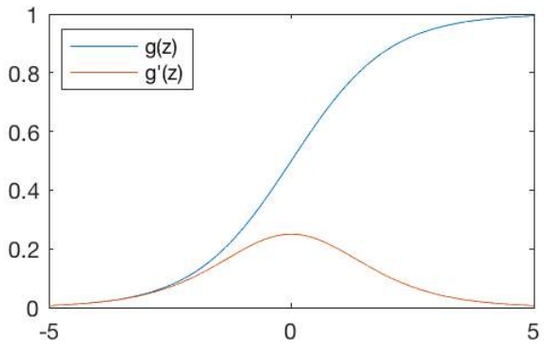


$$\hat{\mathbf{y}} = g^{(2)}\left(\mathbf{W}^{(2)}\ g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{w}_0^{(1)}\right) + \mathbf{w}_0^{(2)}\right)$$

Activation Function

The purpose of activation functions is to introduce non-linearities into the network

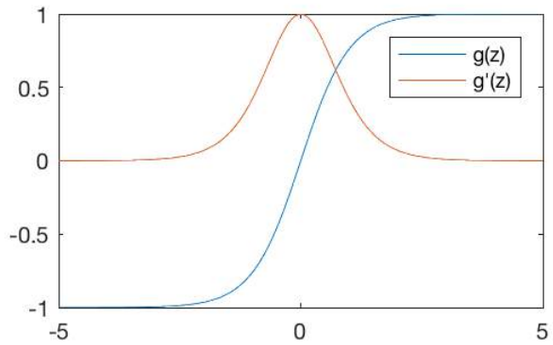
Sigmoid Function



$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

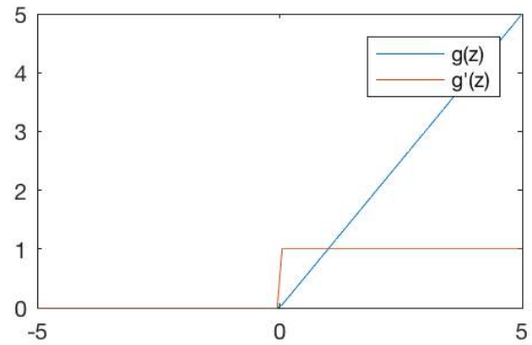
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

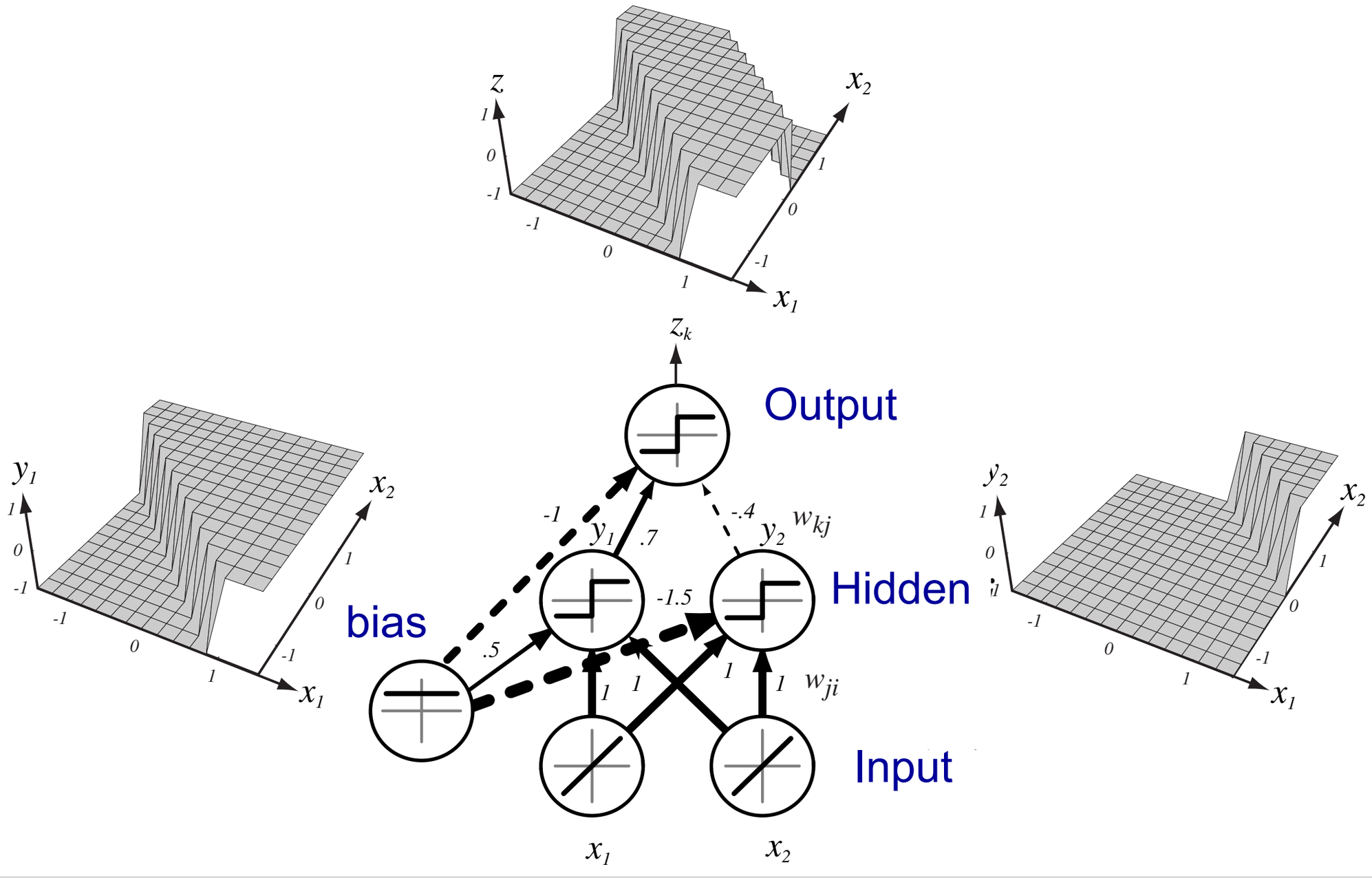
Rectified Linear Unit (ReLU)



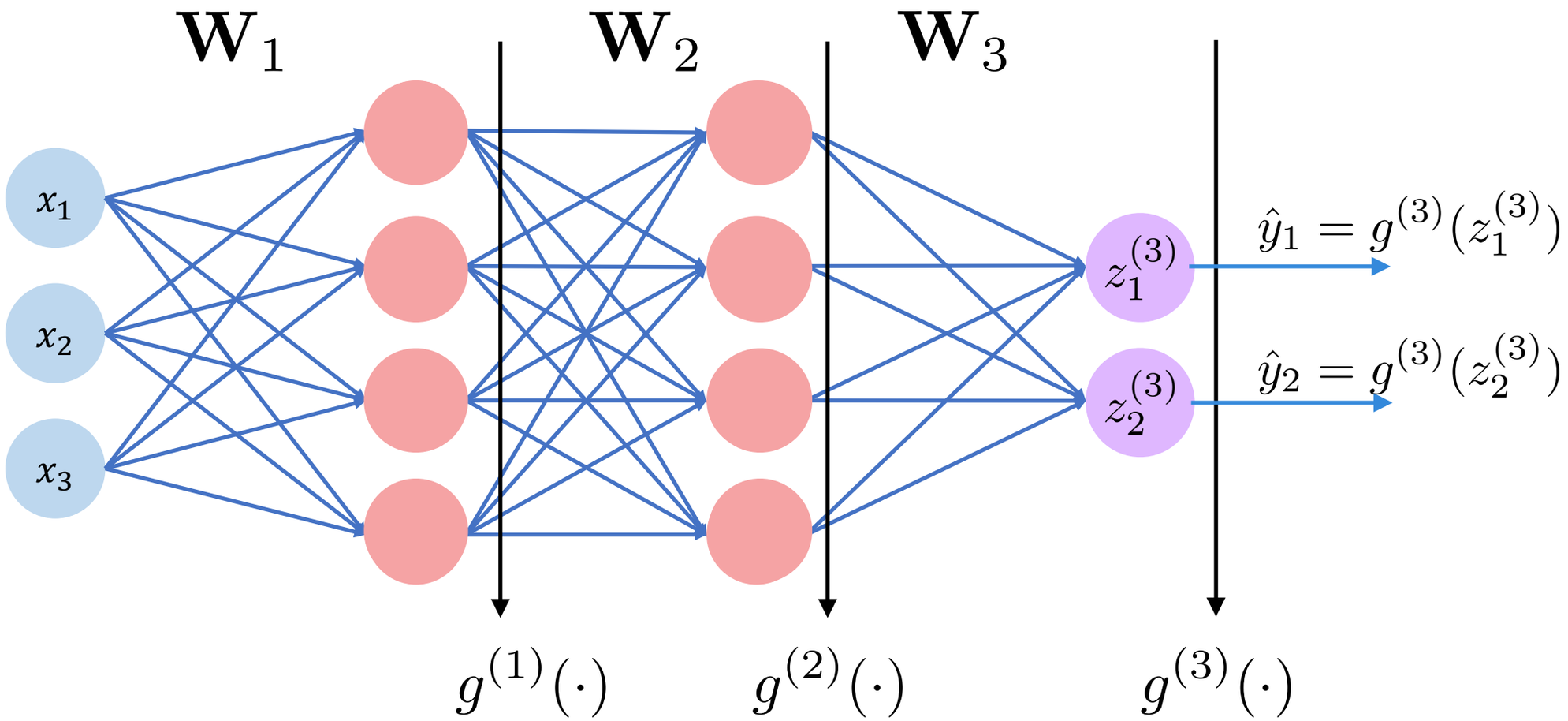
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

A Neural Network with two layers

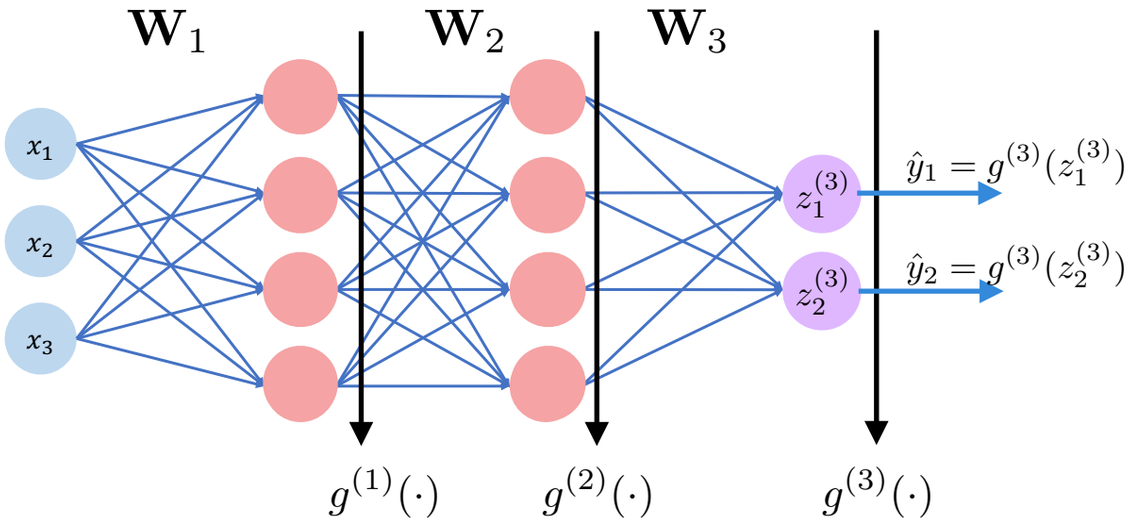


A Neural Network with three layers



$$\hat{\mathbf{y}} = g^{(3)}\left(\mathbf{W}^{(3)}\ g^{(2)}\left(\mathbf{W}^{(2)}\ g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{w}_0^{(1)}\right) + \mathbf{w}_0^{(2)}\right) + \mathbf{w}_0^{(3)}\right)$$

Training Neural Networks (Supervised task)



Parameter Set

$$\mathbf{W} = \left\{ \mathbf{W}^{(i)}, \mathbf{w}_0^{(i)} \right\}_{i=1}^{3}$$

Empirical Loss

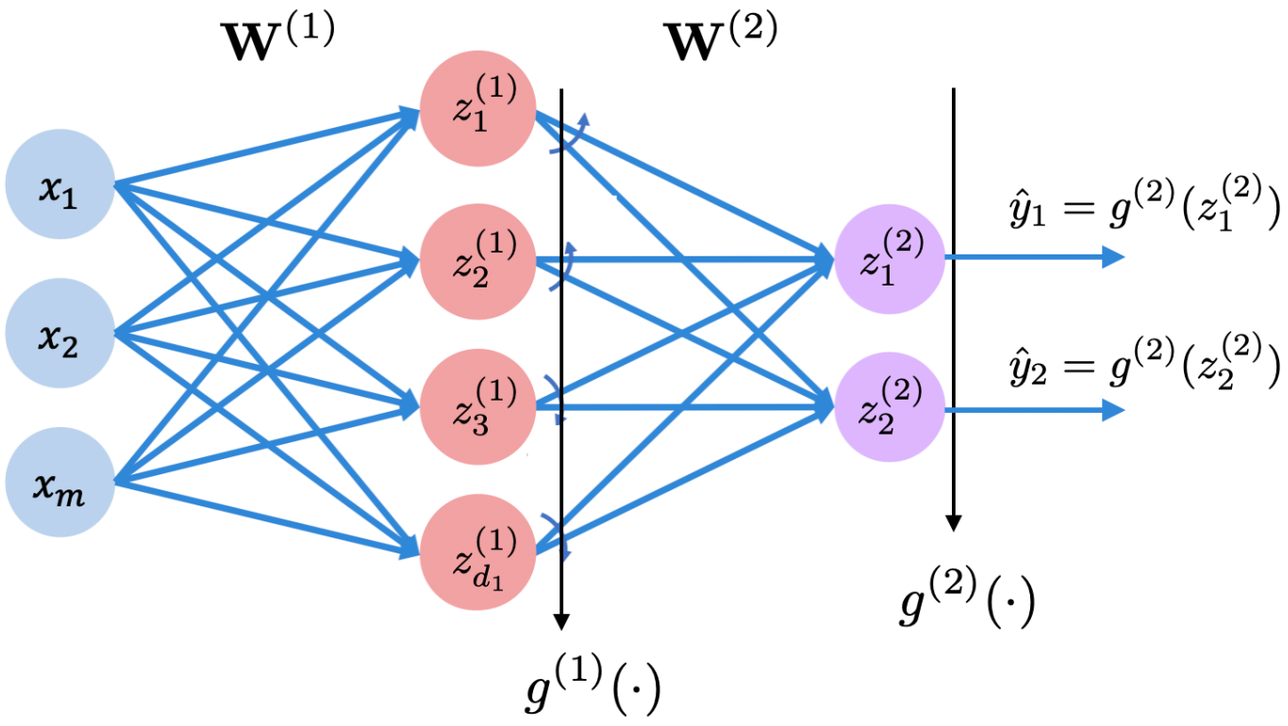
$$J(\mathbf{W}) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)} \right)$$

Model Training

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} J(\mathbf{W})$$

Loss Function (Supervised task)

Multi-output Regression $\mathcal{D} = \left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)_{i=1}^{N} \quad \mathbf{x}^{(i)} \in \mathbb{R}^m \qquad \mathbf{y}^{(i)} \in \mathbb{R}^d$



Squared loss

$$\mathcal{L}\left(\hat{\mathbf{y}}, \mathbf{y}\right) = ||\mathbf{y} - \hat{\mathbf{y}}||^2$$

Linear activation

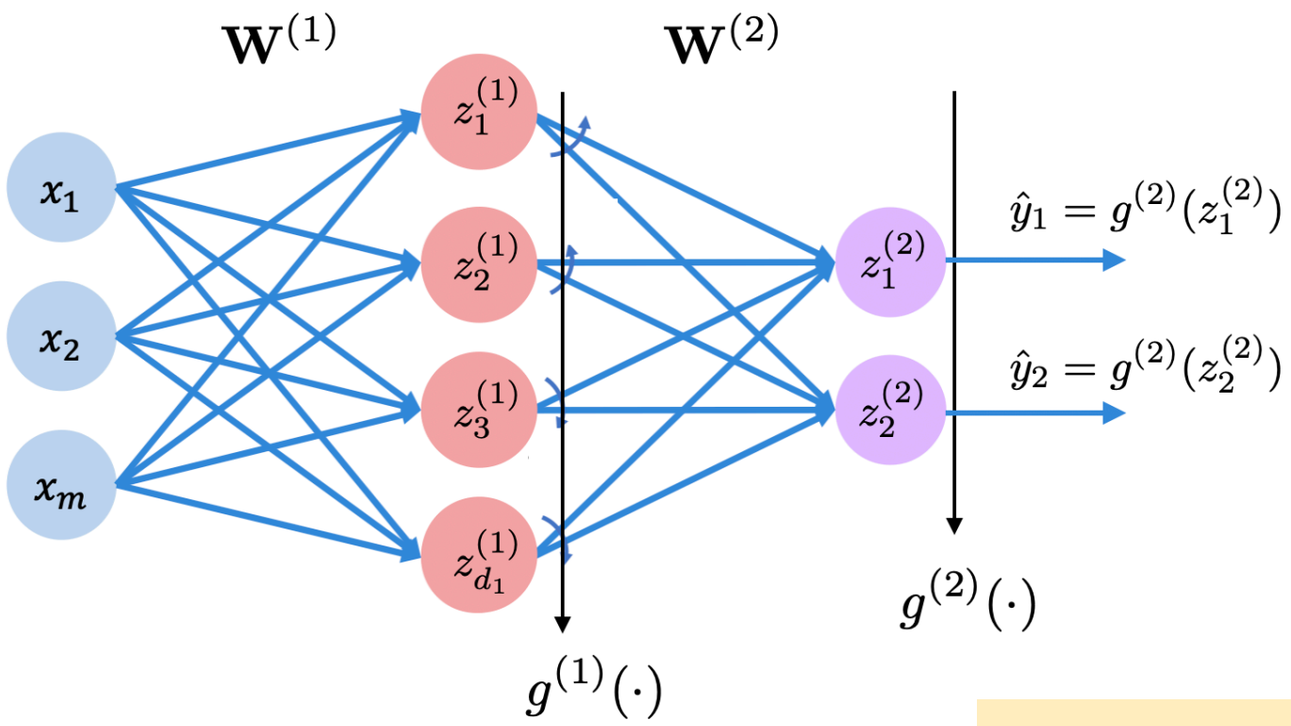
$$g^{(2)}(z) = z$$

What if the output y is positive-real valued?

Loss Function (Supervised task)

Classification $\mathcal{D} \doteq (\boldsymbol{x}^{(i)}, y^{(i)})_{i=1}^{N} \qquad \boldsymbol{x}^{(i)} \in \mathbb{R}^m \qquad y^{(i)} \in \{1, \ldots, K\}$

Convert the network output to a probability distribution with the **softmax function**



Softmax

$$\hat{y}_k = g^{(2)}(z_k^{(2)}) = P(y = k|\boldsymbol{x}) = \frac{\mathrm{e}^{z_k^{(2)}}}{\sum_{j=1}^{K} \mathrm{e}^{z_j^{(2)}}}$$

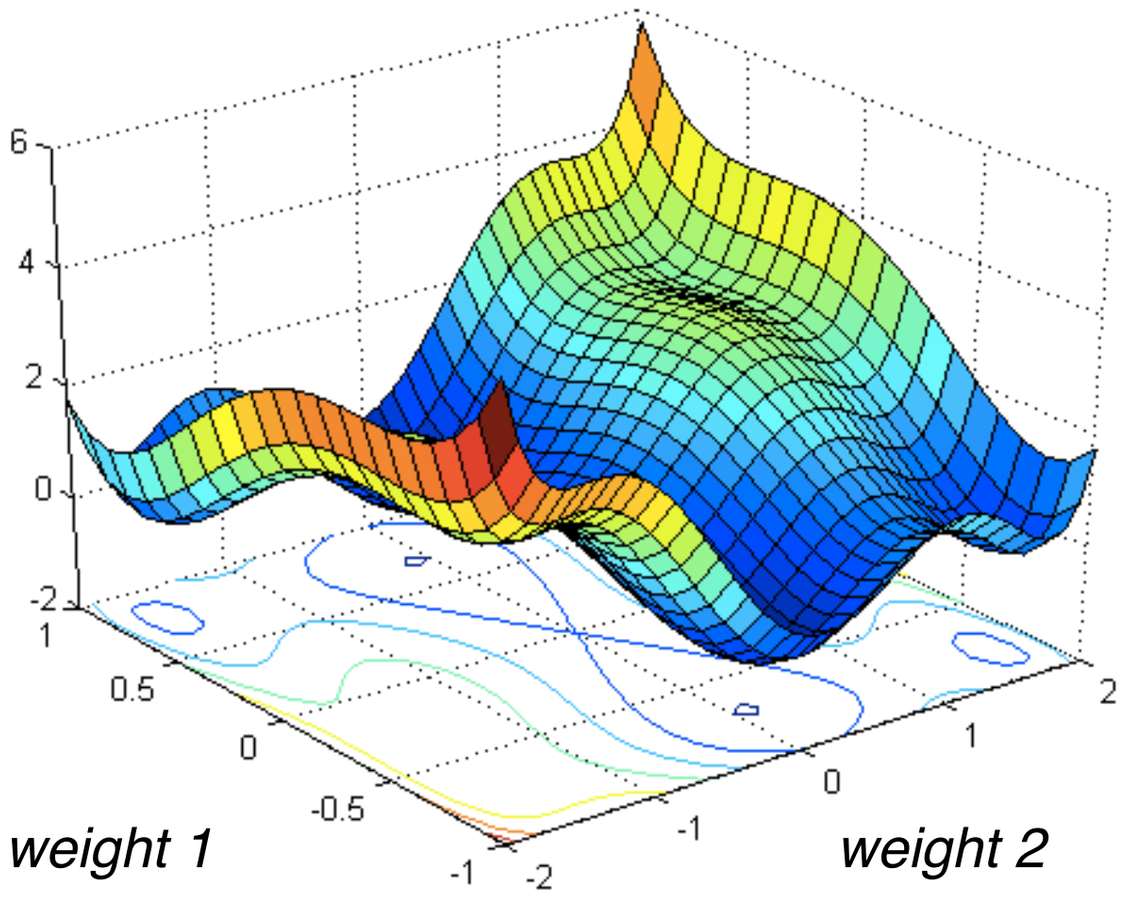
Cross Entropy Loss Function

$$\mathcal{L}(\hat{\mathbf{y}}, y) = -\sum_{k=1}^{K} \mathbb{I}[y == k] \log P(y = k|\boldsymbol{x})$$

Training NNs

uc3m

Optimization is highly non-convex …

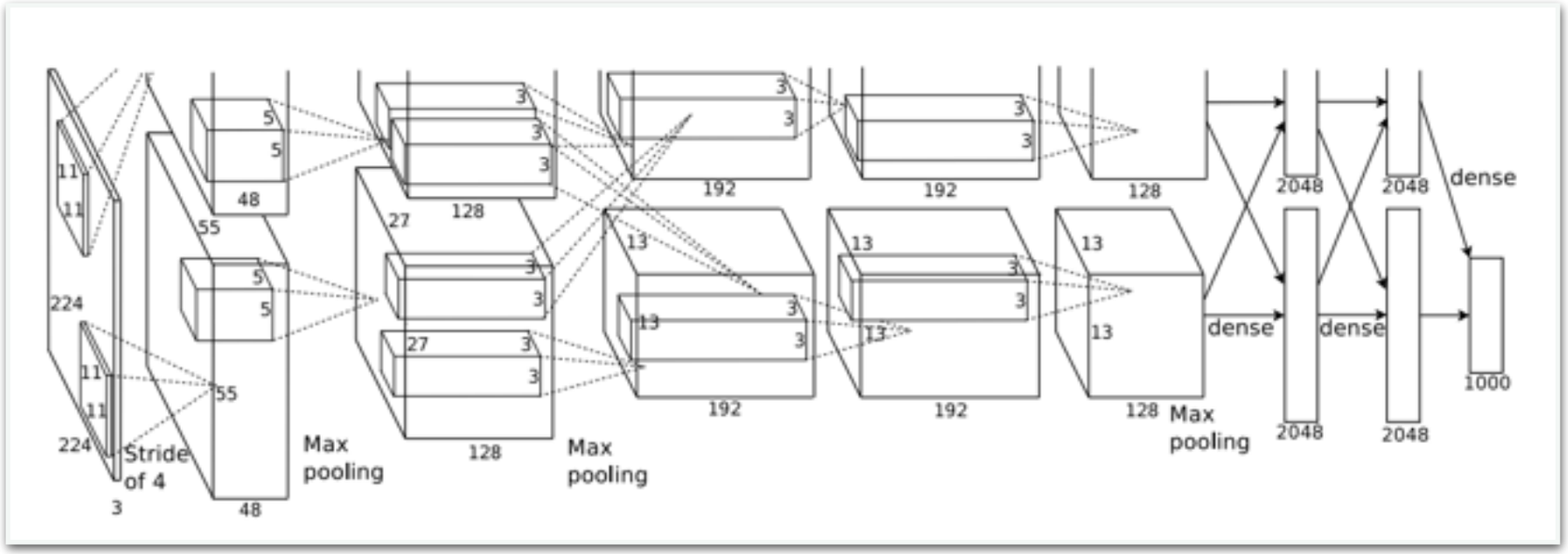


$$\mathbf{W}^* = \arg\min_{\mathbf{W}} J(\mathbf{W})$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right)$$

Note that deep networks operation in O(1M) dimensions!

Brutally simple but engineering matters



ImageNet Classification with Deep Convolutional Neural Networks

A Krizhevsky I Sutskever, G Hinton (2012)

- Deep CNN (AlexNet) won 2012 ImageNet classification contest
- 60 Millions Parameters
- **Exploit data correlation**
- **Achieving scale in compute and data** is critical

… but training is brutally simple (we assume we never get to the global optima)

Gradient Descent

Algorithm

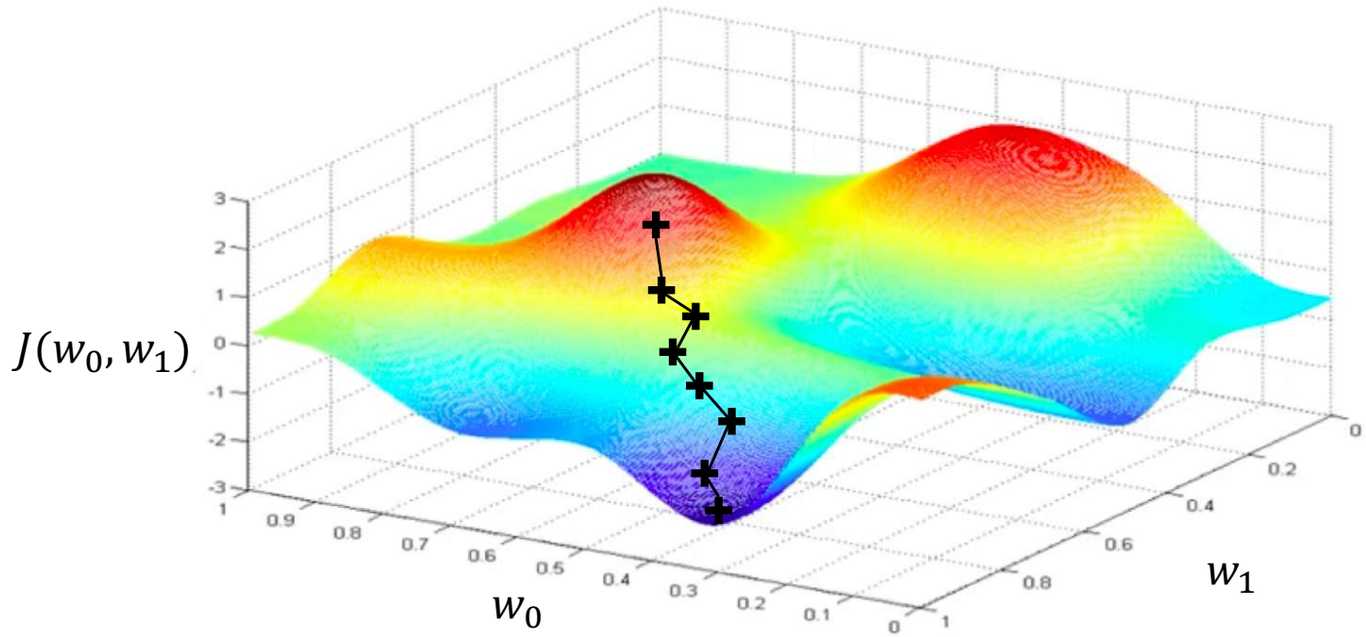
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

   3. Compute gradient $\quad \frac{\partial J}{\partial \mathbf{W}}$

   4. Update Weights $\quad \mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J}{\partial \mathbf{W}}$

5. Return weights



Check out this excellent post about initialization!

… but training is brutally simple (we assume we never get to the global optima)

Gradient Descent

Algorithm

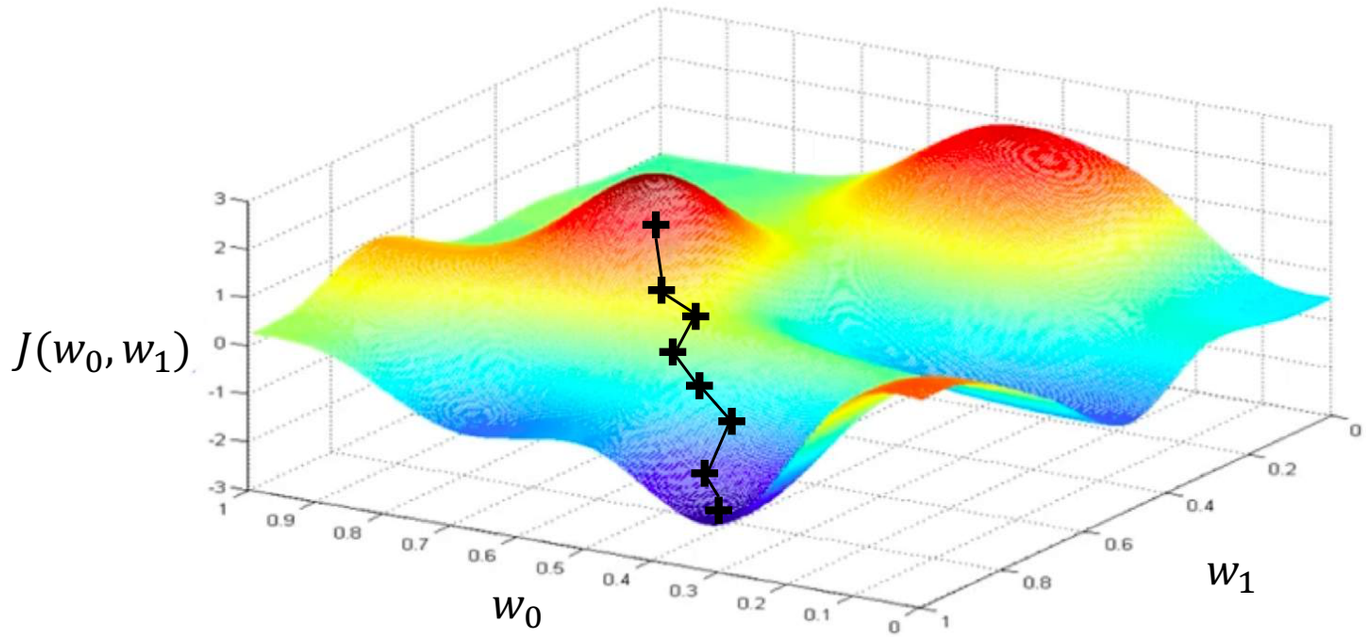
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

   3. Compute gradient $\frac{\partial J}{\partial \mathbf{W}}$

   4. Update Weights $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J}{\partial \mathbf{W}}$

5. Return weights



Computing gradients efficiently: backpropagation

- A deep NN can contain millions of parameters

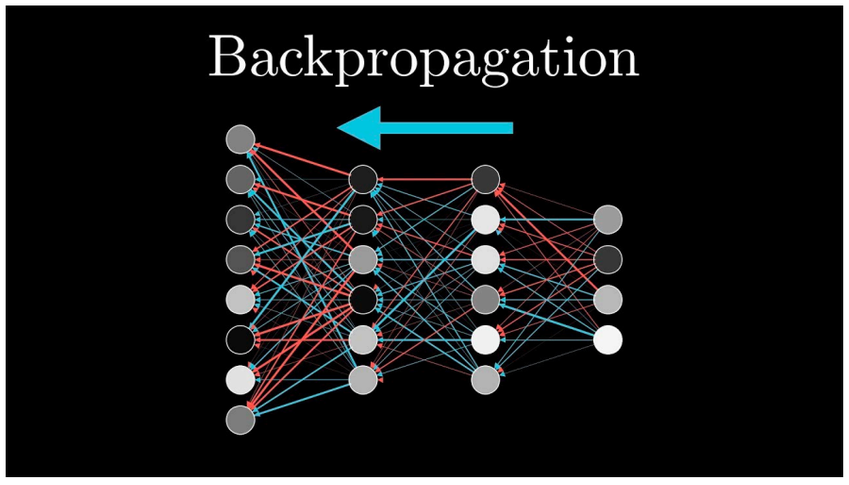
- Gradients cannot be computed individually for parameter, there must be some way of "reusing" computations

- **Backpropagation** is the key to understand why NNs have such a particular structure:

  - A multi-variate linear operator (efficiently implemented in HW)
  - Followed by element-wise non-linear functions

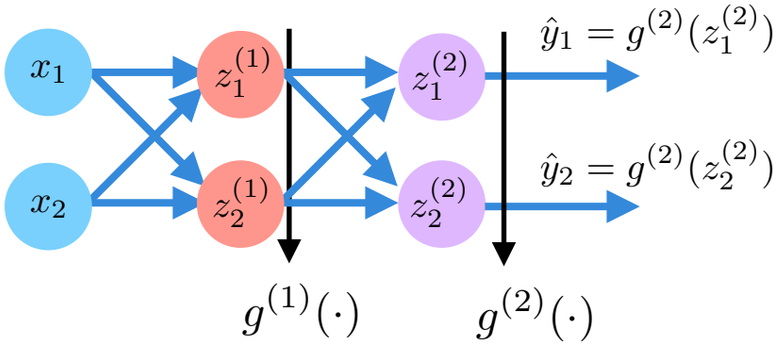
- Thanks to this structure, with **Backpropagation** we evaluate gradients from the top (output) of the network to the bottom (input) by reusing computations.

- Cost of computing gradients ~ cost of evaluating the NN output for a given input



Example for a 2 layer NN with linear outputs (regression)

- $g^{(2)}(z) = z \qquad \mathcal{L}_n = ||\mathbf{y}_n - \hat{\mathbf{y}}_n||^2 \qquad \delta_i^{(2)} = \dfrac{\partial \mathcal{L}_n}{\partial z_i^{(2)}} = -2(y_1 - z_i^{(2)}), \quad i = 1, 2$



- $z_i^{(2)} = w_{i1}^{(2)} g_1(z_1^{(1)}) + w_{i2}^{(2)} g_1(z_2^{(1)}) + w_{0,i}^{(2)}$
- $\dfrac{\partial \mathcal{L}_n}{\partial w_{i1}^{(2)}} = \dfrac{\partial \mathcal{L}_n}{\partial z_i^{(2)}} \dfrac{z_i^{(2)}}{\partial w_{i1}^{(2)}} = \delta_i^{(2)} g_1(z_1^{(1)})$
- $\dfrac{\partial \mathcal{L}_n}{\partial w_{i2}^{(2)}} = \dfrac{\partial \mathcal{L}_n}{\partial z_i^{(2)}} \dfrac{z_i^{(2)}}{\partial w_{i2}^{(2)}} = \delta_i^{(2)} g_1(z_2^{(1)})$

- $\dfrac{\partial \mathcal{L}_n}{\partial z_i^{(1)}} = \delta_i^{(1)} = \dfrac{\partial \mathcal{L}_n}{\partial z_1^{(2)}} \dfrac{\partial z_1^{(2)}}{\partial z_i^{(1)}} + \dfrac{\partial \mathcal{L}_n}{\partial z_2^{(2)}} \dfrac{\partial z_2^{(2)}}{\partial z_i^{(1)}} = \delta_1^{(2)} w_{1i}^{(2)} g_1'(z_i^{(1)}) + \delta_2^{(2)} w_{2i}^{(2)} g_1'(z_i^{(1)}) = g_1'(z_i^{(1)}) \left(\delta_1^{(2)} w_{1i}^{(2)} + \delta_2^{(2)} w_{2i}^{(2)} \right)$

Backpropagation general formula

$$\delta_j^{(m-1)} = g'(z_j^{(m-1)}) \sum_{k=1}^{d_m} w_{k,j} \delta_k^{(m)}$$

Stochastic Backpropagation

Assume an M-layer MLP

$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} \longrightarrow \mathbf{g}^{(1)} = g(\mathbf{z}^{(1)}) \longrightarrow \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{g}^{(1)} \longrightarrow \cdots \longrightarrow \hat{\mathbf{y}} = g(\mathbf{z}^{(M)})$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) = \sum_{n=1}^{N} \mathcal{L}_n \longrightarrow \frac{\partial J(\mathbf{W})}{\partial w_{ji}^{(m)}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}}$$

Very costly for massive datasets!

Stochastic Gradient Descent (SGD)

Select at random a mini batch $\mathcal{B}$ of data at every SGD iteration

$$\frac{\partial J(\mathbf{W})}{\partial w_{ji}^{(m)}} \approx \sum_{n \in \mathcal{B}} \frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}}$$

Parallelizable!
Very efficient in GPUs

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim^{\text{iid}} \mathcal{N}(0, \sigma^2$

2. Loop until convergence:

3. Compute (noisy) gradient $\frac{\partial J}{\partial \mathbf{W}}$

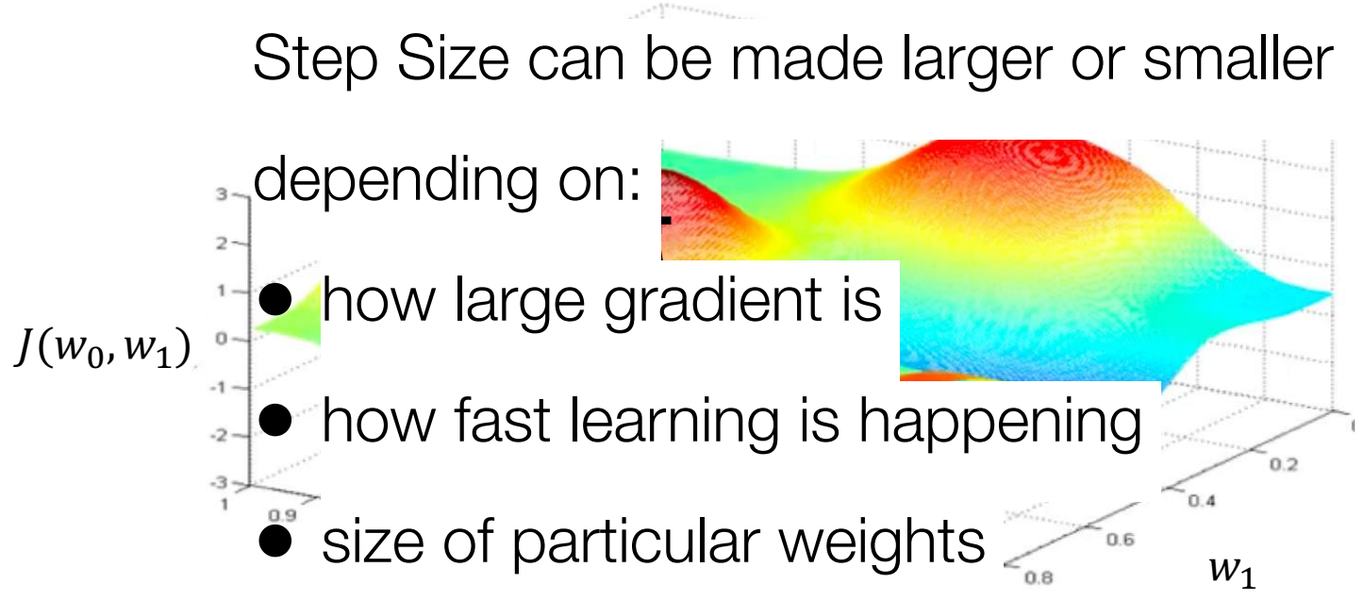
4. Update Weights $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J}{\partial \mathbf{W}}$

5. Return weights

Adaptive Learning Rates

Step Size can be made larger or smaller depending on:

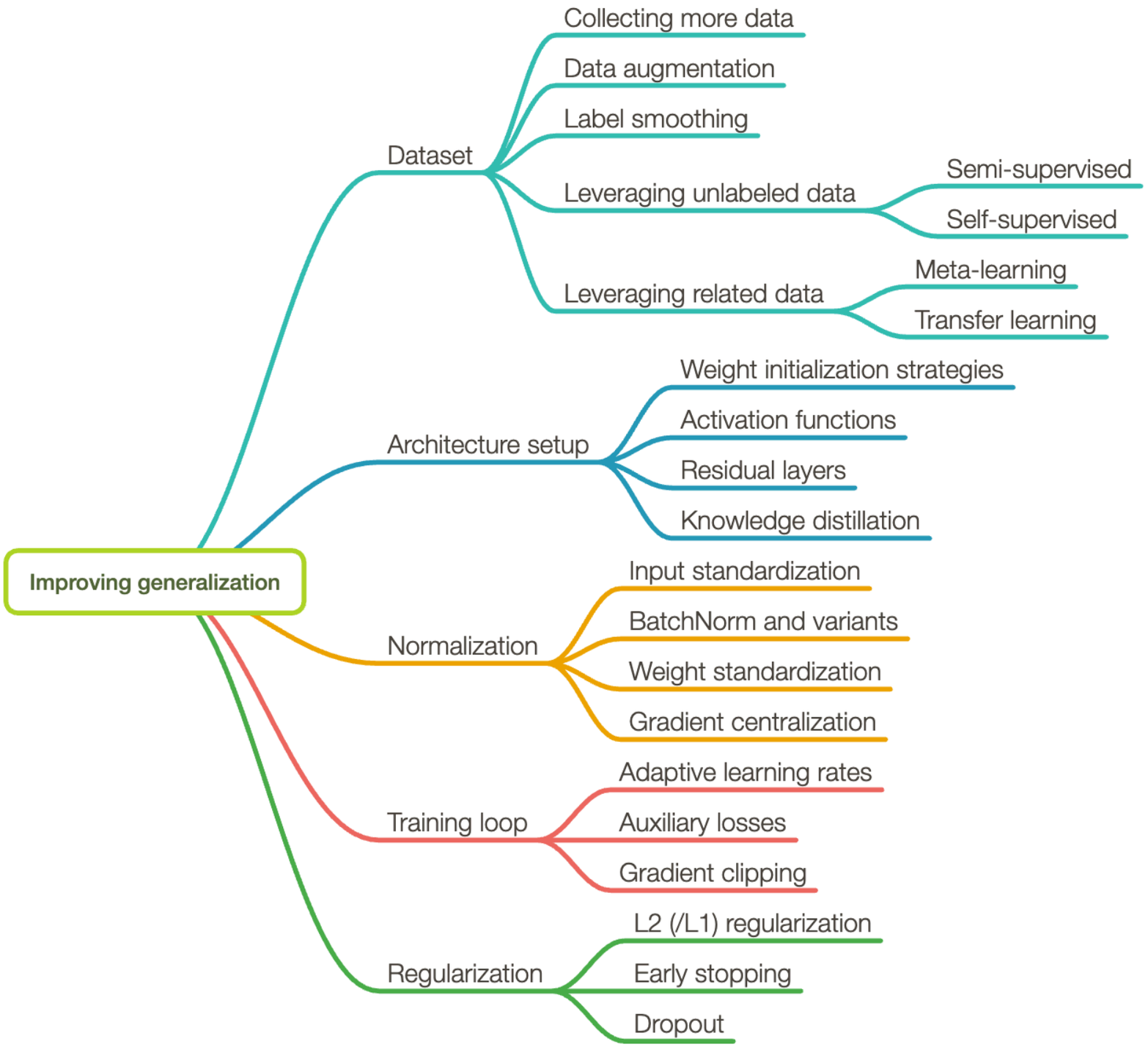
- how large gradient is
- how fast learning is happening
- size of particular weights
- etc...



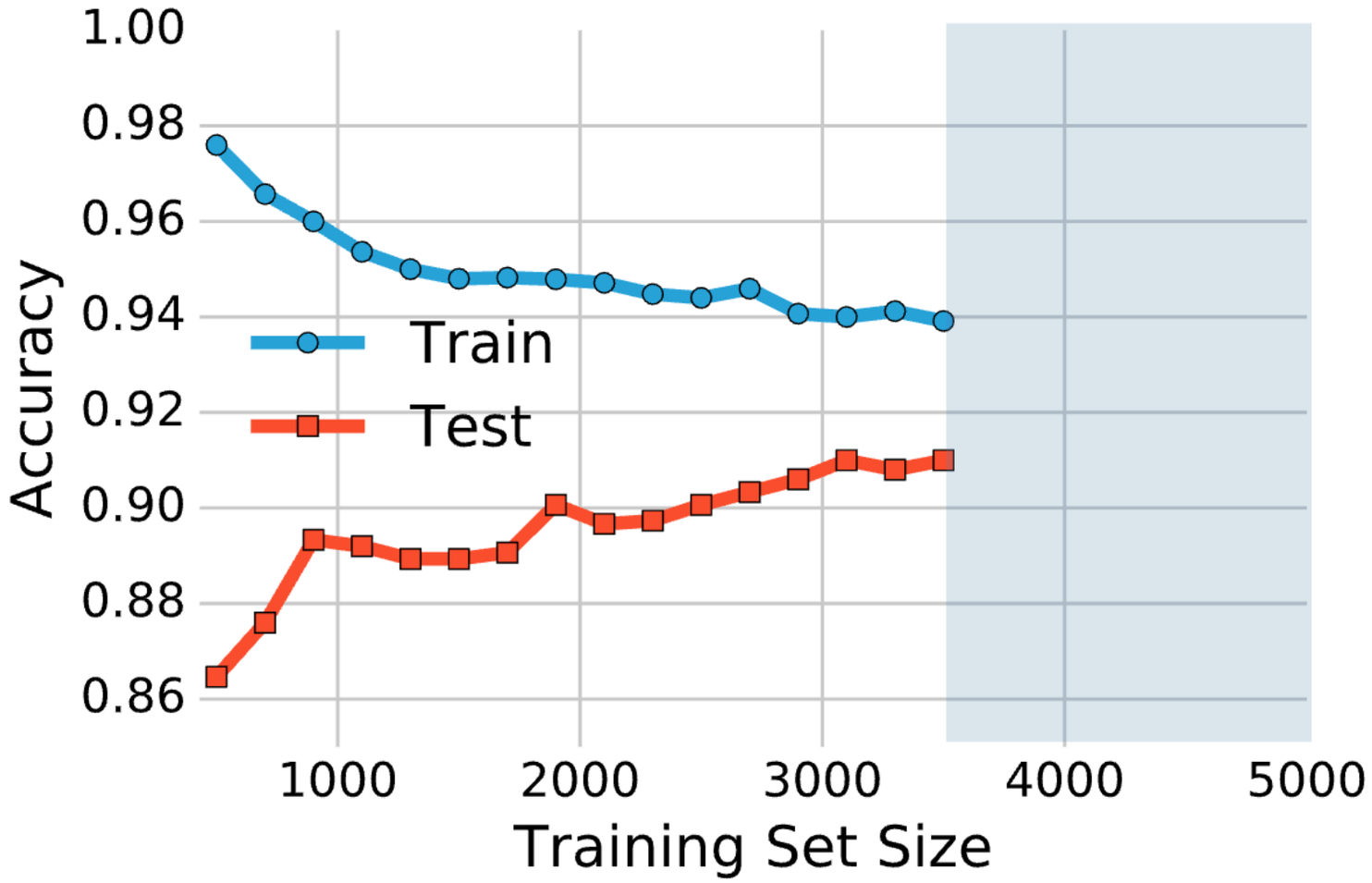
Momentum, Adagrad, Adam, RMSProp, … (Check out this post)

How to improve generalization?

Reduce Overfitting & Improve Regularization Performance



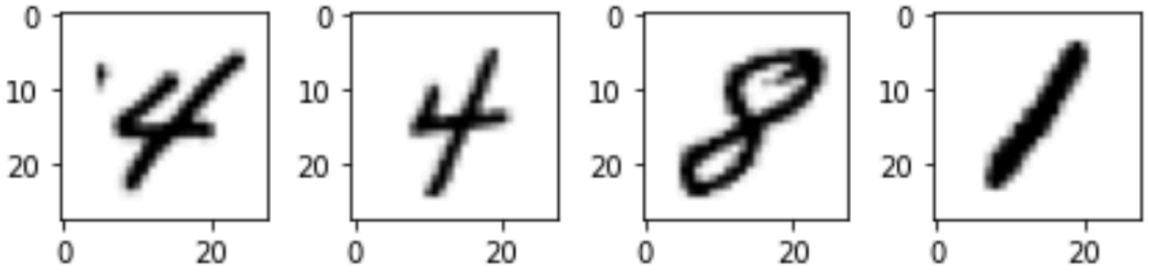
Avoid Overfitting with More Data



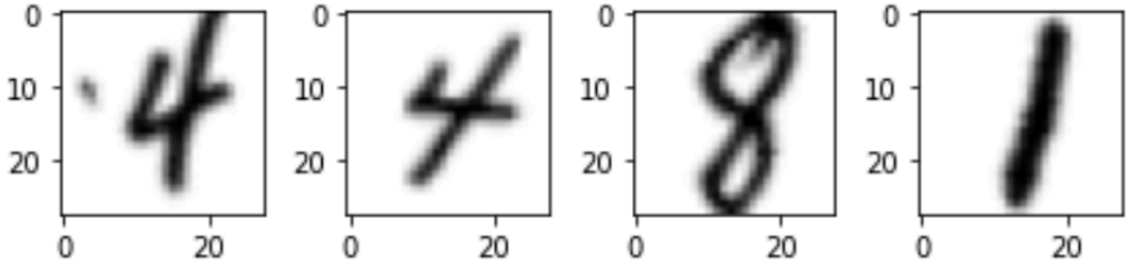
Softmax on MNIST subset (test set size is kept constant)

Avoid Overfitting with Data Augmentation

Original



Randomly augmented



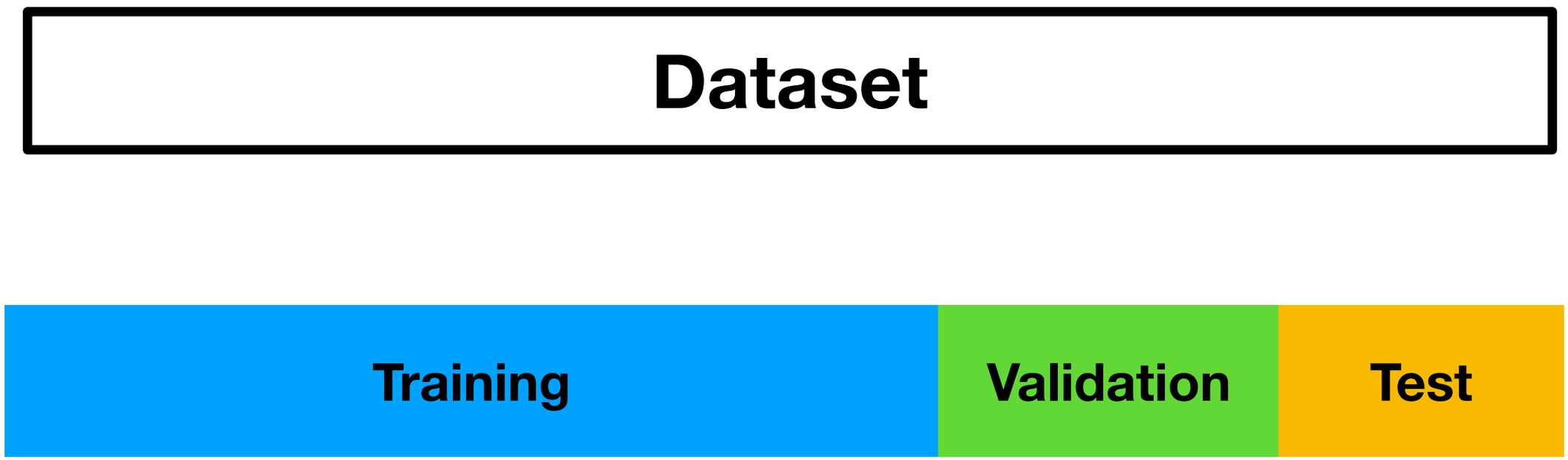
Link to Code Example

sebastianraschka.com

Regularization: Early Stopping

Remember to split dataset into (training, validation, test)

- Use validation accuracy for tuning
- Use test only at the end (when we have selected our best model)



Model Selection

- Train your model on the **training set** $\mathcal{D}_{\text{train}}$

- For model selection, use a **validation set** $\mathcal{D}_{\text{val}}$

  ✓Hyper-parameter search: hidden layer sizes, number of layers, learning rate, number of iterations/epochs, …

- Estimate generalization performance using a **test set** $\mathcal{D}_{\text{test}}$

Regularization

Regularization Goal: Reduce Overfitting

- Usually achieved by reducing model capacity and/or reducing the variance of preditions
- **Common regularization techniques for DNNs:**

✓Early Stopping

✓L1/L2 regularization (norm penalties)

✓Dropout

Regularization: Early Stopping

To select the number of epochs, stop training **when validation set error increases** (with some look ahead)

L1 & L2 Regularization

A “weight shrinkage” or a “penalty against complexity”

- L1 regularization => LASSO Regression
- L2 regularization => Ridge Regression

Let’s focus on L2 regularization:

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right)$$

hyperparameter

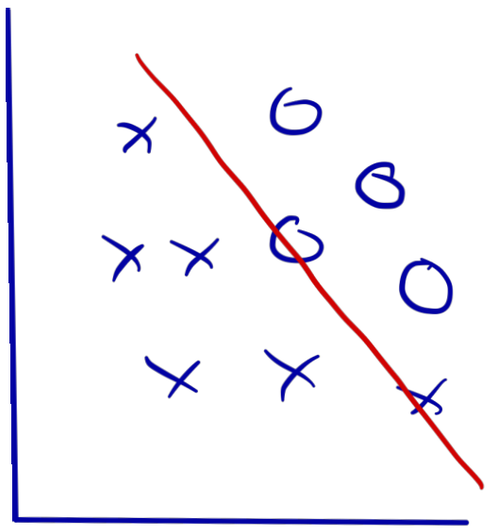
L2-regularization:

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) + \frac{\lambda}{N}\sum_{j}\omega_j^2$$

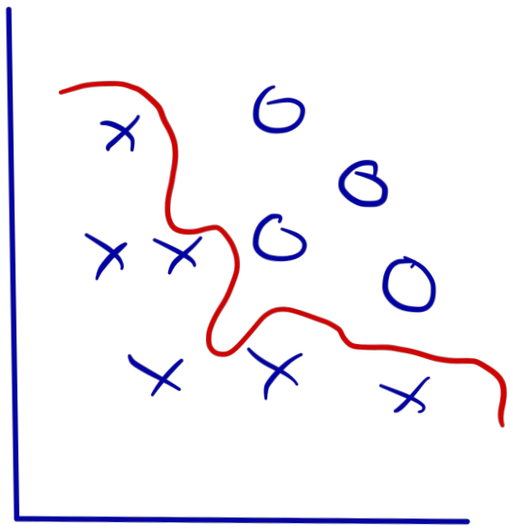
where: $\sum_{j}\omega_j^2 = \|\mathbf{w}\|_2^2$

L2 Regularization Effect on Decission Boundaries

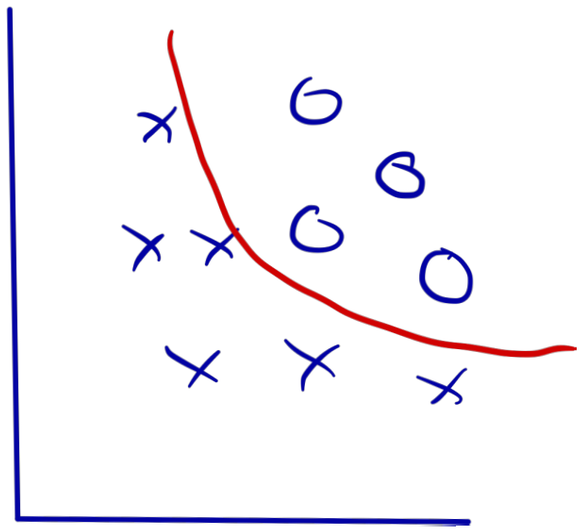
1. Large regularization penalty => high bias
2. Low regularization penalty => high variance
3. Good compromise



1.



2.



3.

L2 Regularization in Neural Networks

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) + \frac{\lambda}{N}\sum_{l=1}^{L}\|\mathbf{w}^{\mathbf{l}}\|_F^2$$

where $\|\mathbf{w}^l\|_F^2$ is the Frobenius norm (squared)

sum over layers

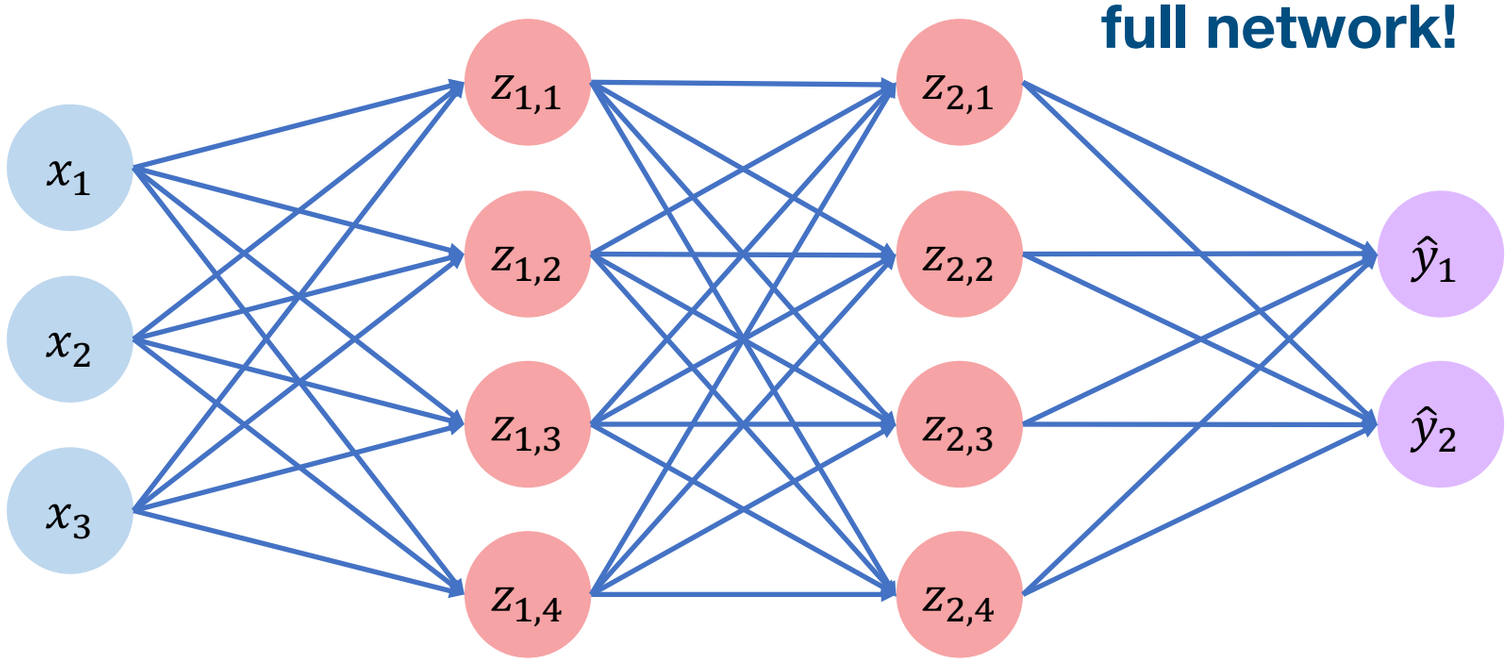
$$\|\mathbf{w}^l\|_F^2 = \sum_i \sum_j (\omega_{i,j}^l)^2$$

Regularization: Dropout

During training cripple neural network by **removing hidden units stochastically**

- Randomly set some activations to 0
- Typically ‘drop’ 50% of activations in layer
- Forces network to not rely on any 1 node

In validation & test we use the full network!



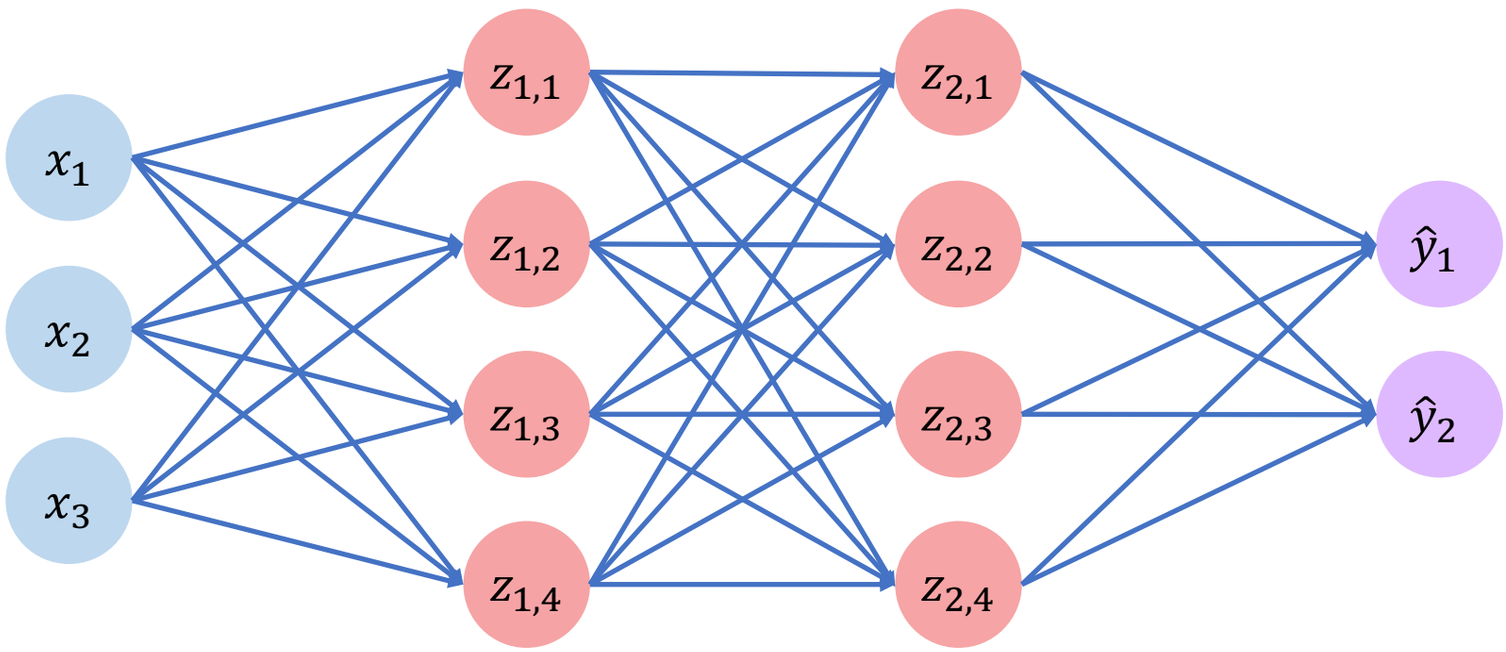
Dropout: Possible Interpretation

1. Network will learn **not to rely on specific connections** too heavily
   - We'll consider more connections
   - Weight values more "spread-out" (similar to L2 norm)
2. **Ensemble method interpretation**:
   - Different model for each minibatch (weight-sharing => regularizes)
   - Infernce can be seen as a geometric average of those models

Normalization Layers

Internal shift

- Training Deep Neural Networks is **complicated** by the fact that **the distribution of each layer's inputs changes during training**, as the parameters of the previous layers change
- We refer to this phenomenon as **internal covariate shift**
- **This slows down the training** by requiring lower learning rates, and **makes it notoriously hard to train models with saturating nonlinearities**



Batch Normalization

- By **fixing the distribution of the layer inputs** as the training progresses, we expect to improve the training speed
- Batch Normalization performs input normalization in a way that is **differentiable** and **does not require the analysis of the entire training set** after every parameter update
- **Normalize each scalar feature independently**
- Scale the normalized input **with trainable parameters**

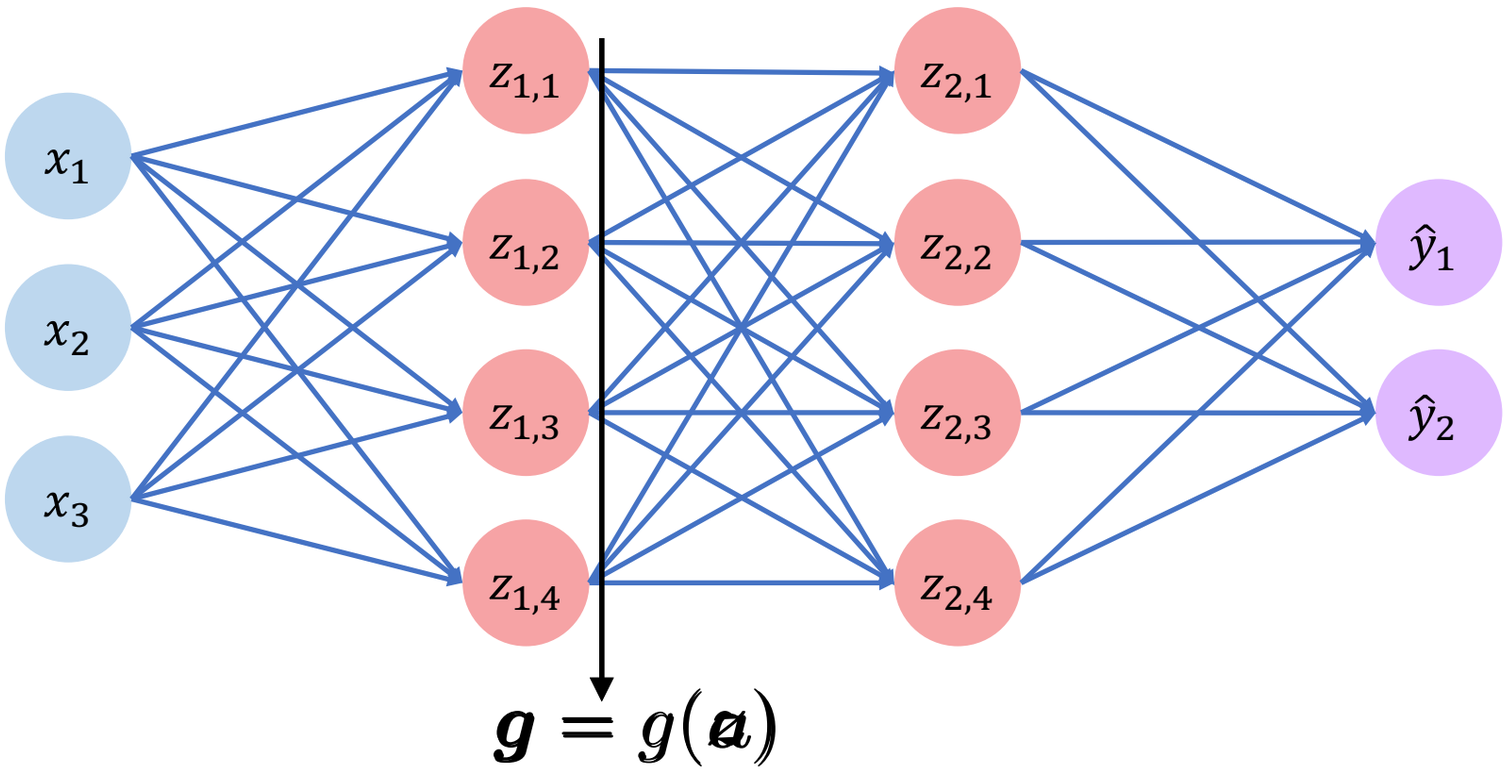
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

---

Sergey Ioffe SIOFFE@GOOGLE.COM
Christian Szegedy SZEGEDY@GOOGLE.COM
Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043

Batch Normalization

- Mini-batch of size M



Input to the non-linear activation

Mini-batch mean

$$\mu_j \leftarrow \frac{1}{M}\sum_{i=1}^{M} z_j^{(i)}$$

Mini-batch variance

$$\sigma_j^2 \leftarrow \frac{1}{M}\sum_{i=1}^{M}(z_j^{(i)} - \mu_j)^2$$

Normalize

$$\hat{z}_j^{(i)} = \frac{z_j - \mu_j}{\sqrt{\sigma_j^2}}$$

Scale and shift

$$a_j^{(i)} = \gamma \hat{z}_j^{(i)} + \beta$$

Batch Normalization

- During evaluation, we normalize according to the **whole validation/test datasets**.
  γ, β parameters are fixed!

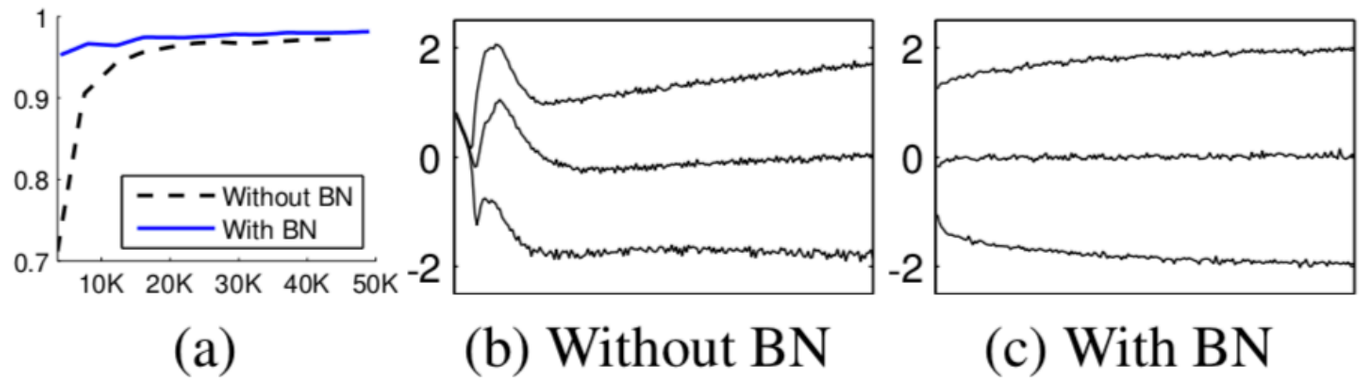


Figure 1. (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Layer Normalization

Layer Normalization

Jimmy Lei Ba
University of Toronto
jimmy@psi.toronto.edu

Jamie Ryan Kiros
University of Toronto
rkiros@cs.toronto.edu

Geoffrey E. Hinton
University of Toronto
and Google Inc.
hinton@cs.toronto.edu

- The effect of batch normalization is dependent on the mini-batch size
- It is not obvious how to apply it to recurrent neural networks.
- In layer normalization, we **normalize the sum of inputs** in a layer using a **single case**.

	Weight matrix re-scaling	Weight matrix re-centering	Weight vector re-scaling	Dataset re-scaling	Dataset re-centering	Single training case re-scaling
Batch norm	Invariant	No	Invariant	Invariant	Invariant	No
Weight norm	Invariant	No	Invariant	No	No	No
Layer norm	Invariant	Invariant	No	Invariant	No	Invariant

Table 1: Invariance properties under the normalization methods.

Layer Normalization

$$a_i^l = w_i^{l\top} h^l \qquad h_i^{l+1} = f(a_i^l + b_i^l)$$

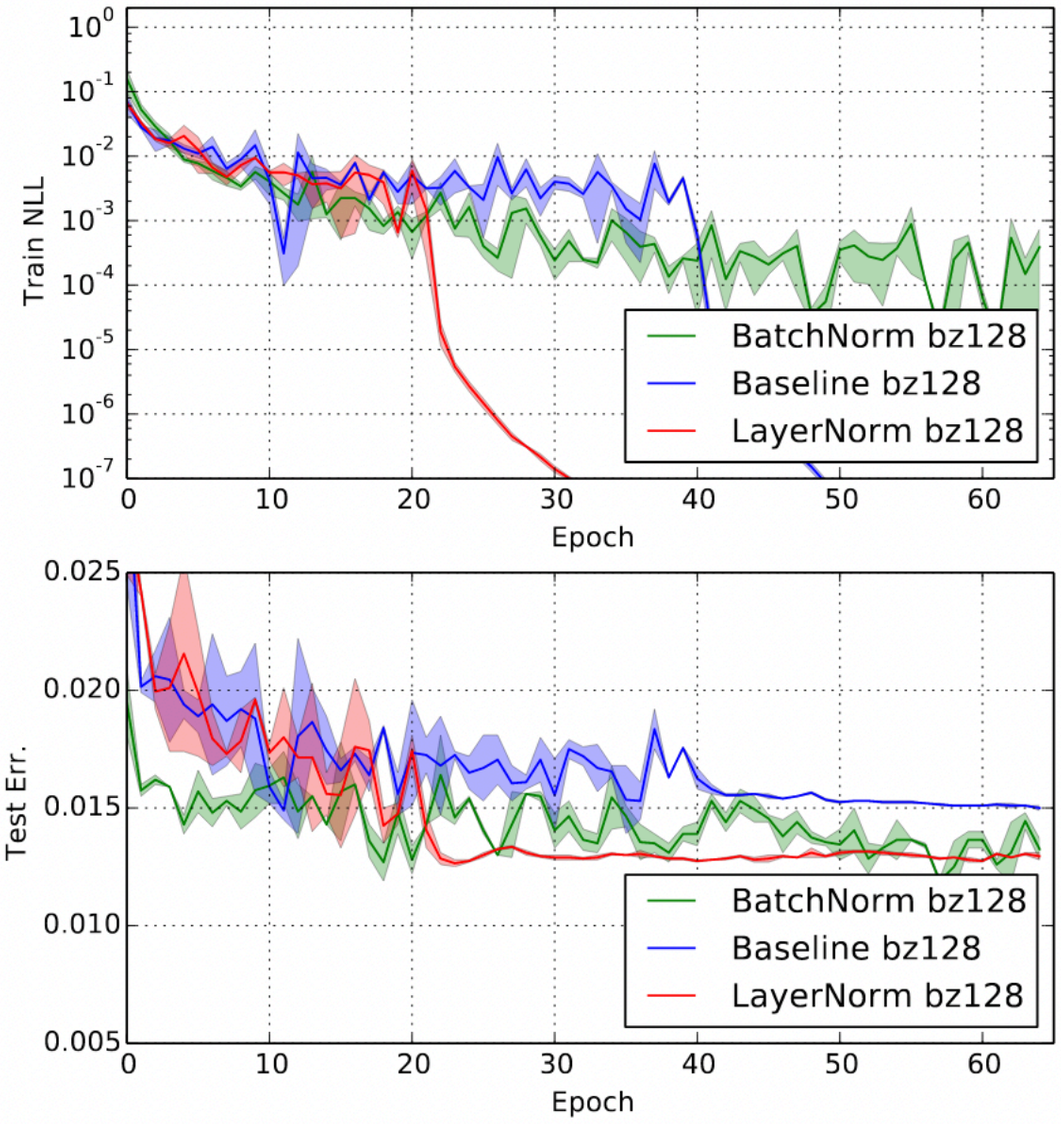
Batch Normalization

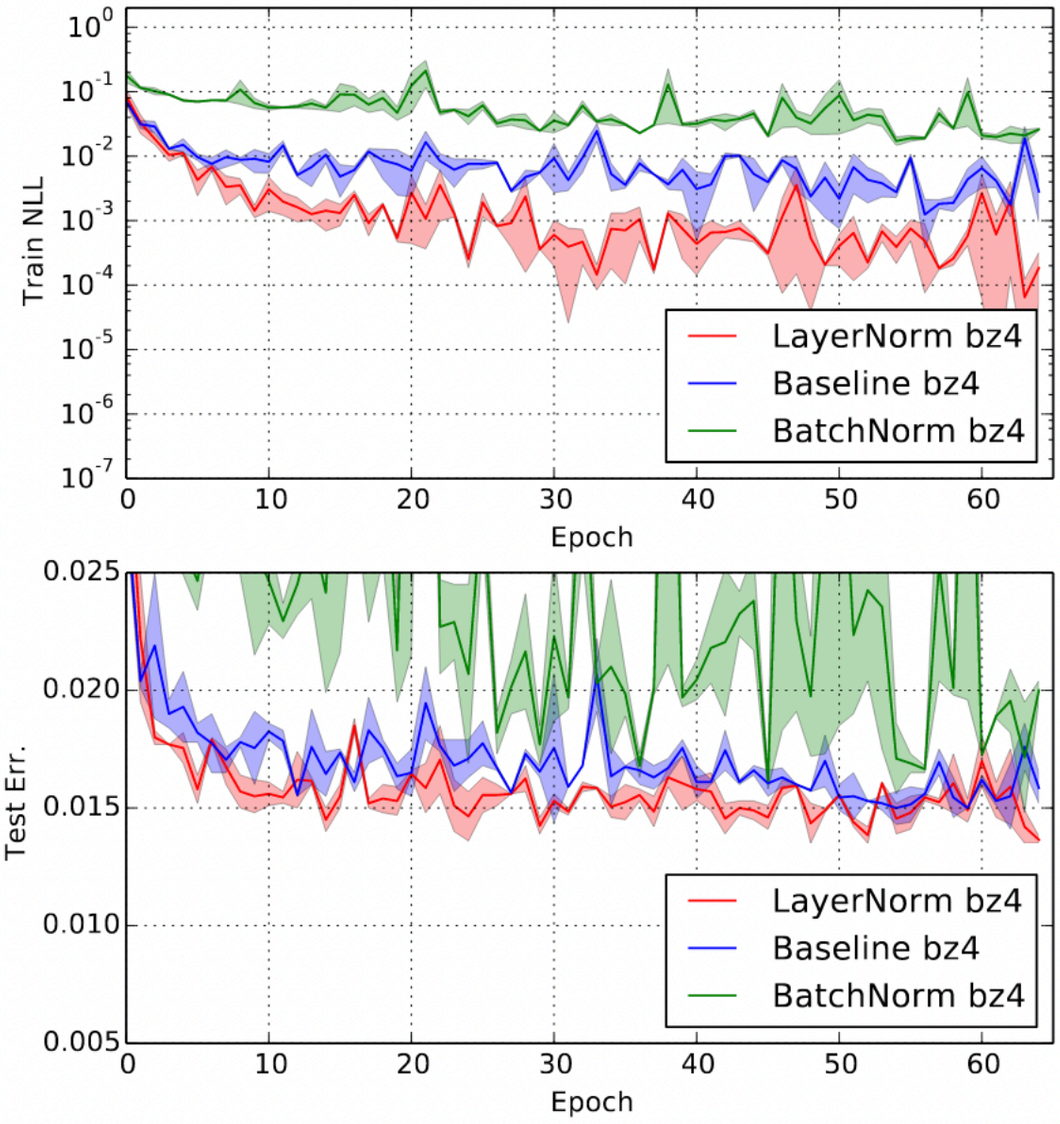
$$\bar{a}_i^l = \frac{g_i^l}{\sigma_i^l}\left(a_i^l - \mu_i^l\right) \qquad \mu_i^l = \mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}\left[a_i^l\right] \qquad \sigma_i^l = \sqrt{\mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}\left[\left(a_i^l - \mu_i^l\right)^2\right]}$$

Layer Normalization

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

Layer Normalization



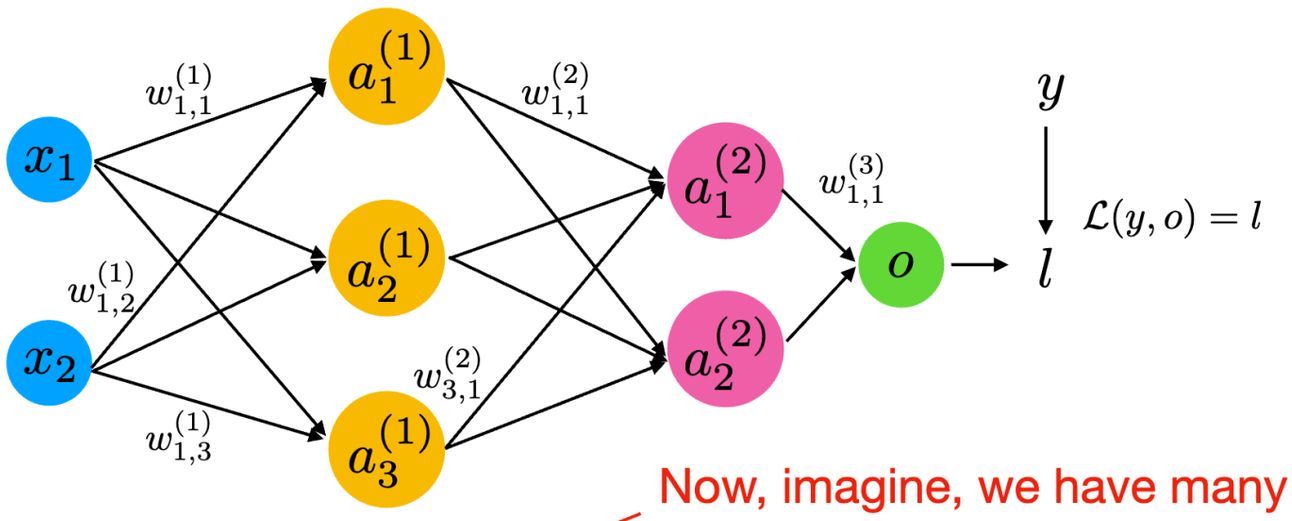


Weight Initialization

Why do we care?

- We have mentioned that we usually initialized weights at random to break symmetry
- Furthermore, we want them to be relatively small but not too much. Why?

✓ **Problem of exploding/vanishing gradients**



Now, imagine, we have many layers and sigmoid activations ...

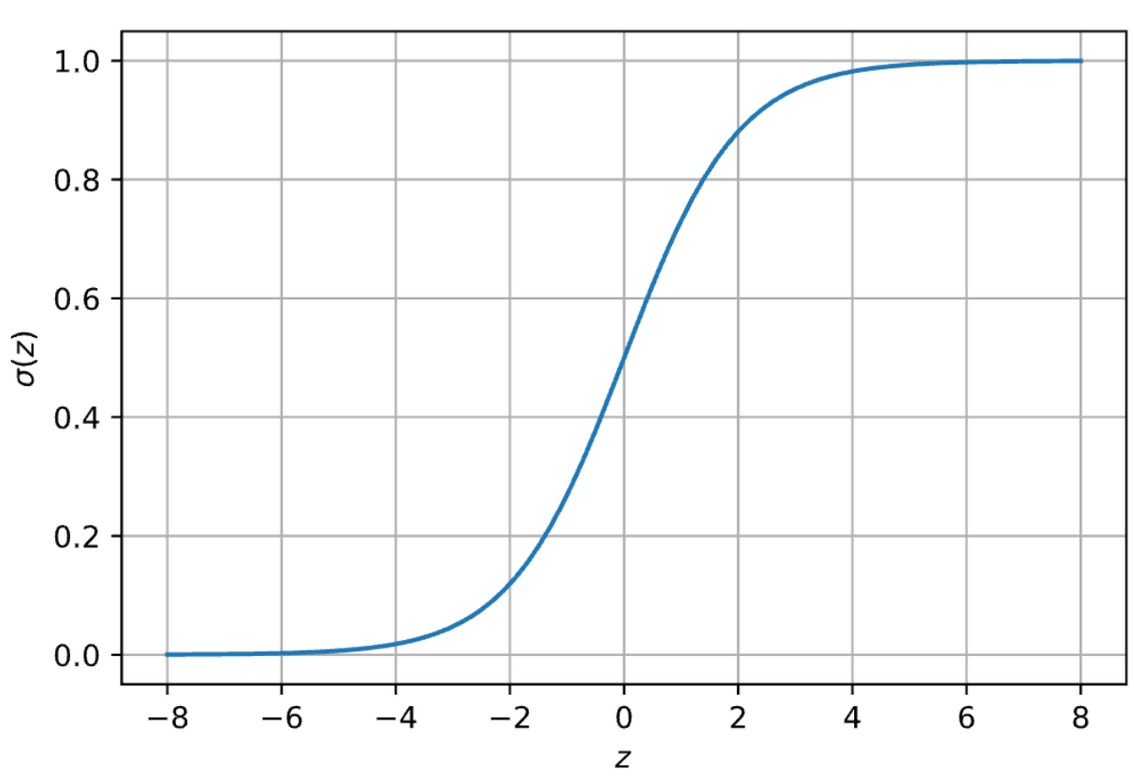
$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

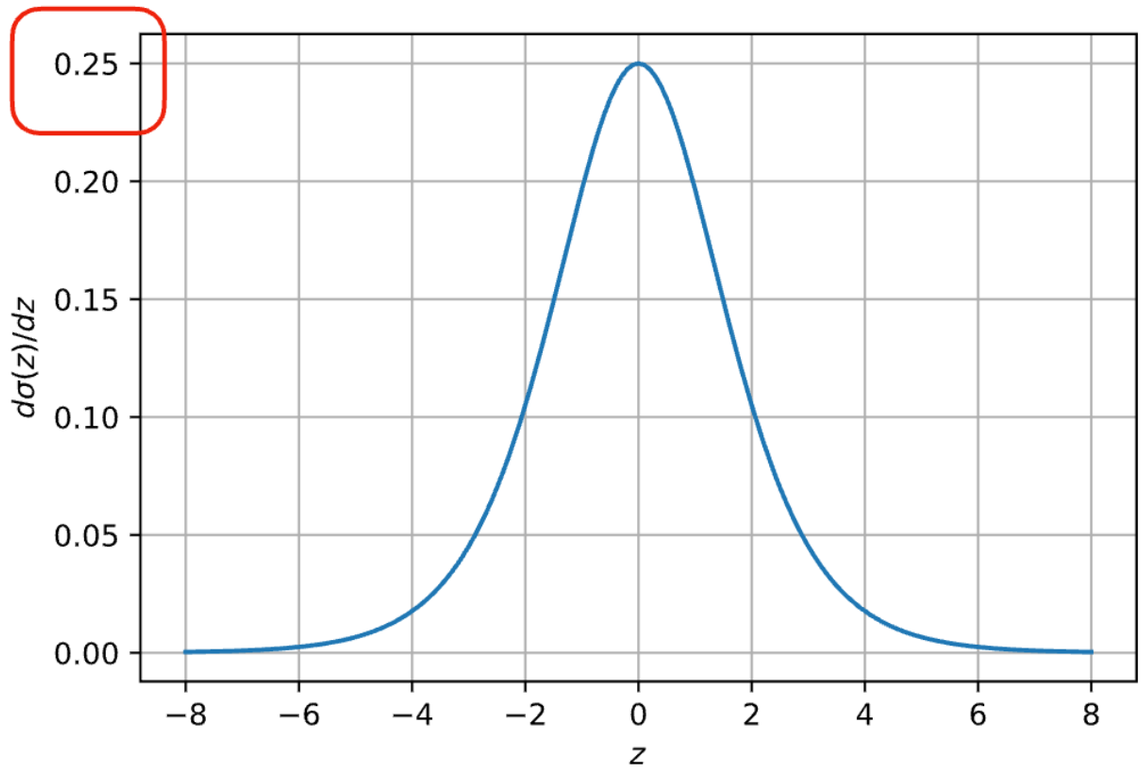
sebastianraschka.com

Vanishing/Exploding Gradient Problems

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

$$\frac{d}{dz}\sigma(z) = \frac{e^{-z}}{(1 + e^{-z})^2} = \sigma(z)(1 - \sigma(z))$$





sebastianraschka.com

Vanishing/Exploding Gradient Problems

Imagine we have the largest gradient of the previous slide

$$\frac{d}{dz}\sigma(0.0) = \sigma(0.0)(1 - \sigma(0.0)) = 0.25$$

If we had 10 layers for instance, as we apply the chain rule to compute gradients, we **degrade the other gradients by orders of magnitude**!

$$0.25^{10} \approx 10^{-6}$$

sebastianraschka.com

Weight Initialization Summary

- Traditionally:
  - ✓ Initialize from a uniform distribution in [0,1] range or [-0.5,0.5]
  - ✓ Sample from a zero-mean Gaussian distribution with small variance (e.g., 0.1 or 0.01)
- More advanced techniques:

✓ TanH function to initialize weights: More robust to vanishing gradients

- Still saturation problem

✓ Xavier Initialization: Small improvement on top of TanH

sebastianraschka.com

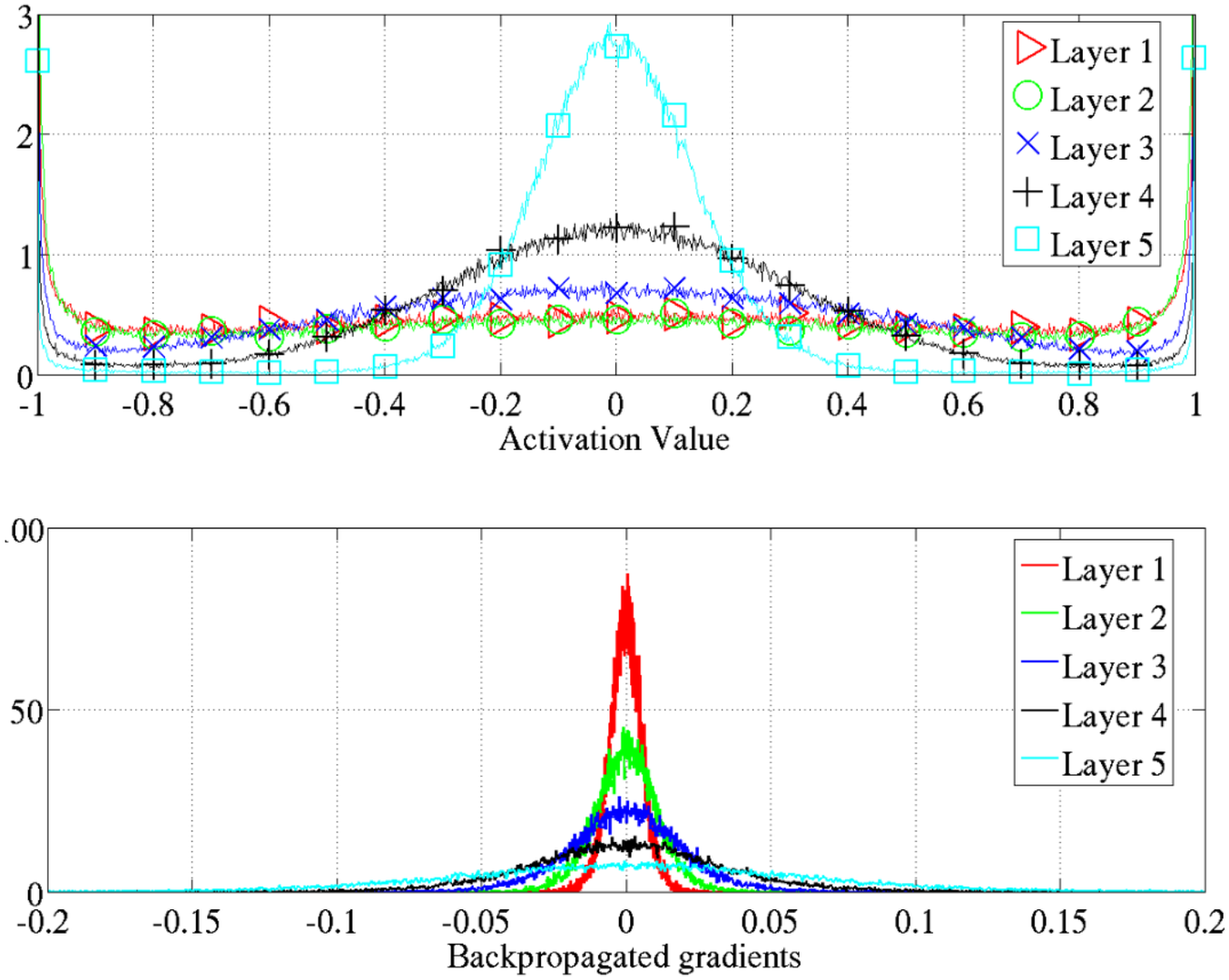
Xavier Initialization

- Initialize Weights from Gaussian or Uniform distribution
- Scale the weights proportional to the number of inputs to the layer
  - ✓For the first hidden layer, that is the number of features in the dataset
  - ✓For the second hidden layer, it is the number of neurons of the first hidden layer
  - ✓Etc.

sebastianraschka.com

Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." in Proc. of the 13th international conference on artificial intelligence and statistics. 2010.



Still has the vanishing gradient problem!

Xavier Initialization

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." in Proc. of the 13th international conference on artificial intelligence and statistics. 2010.

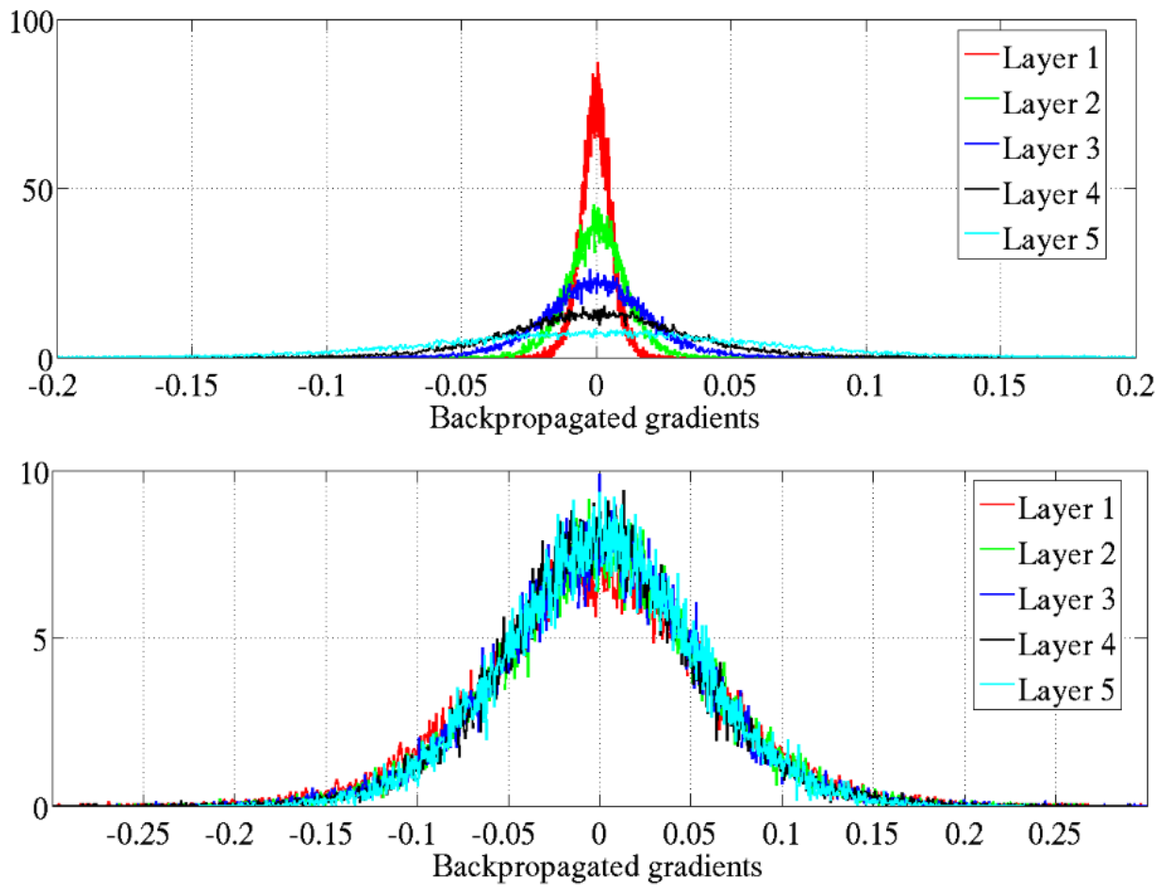


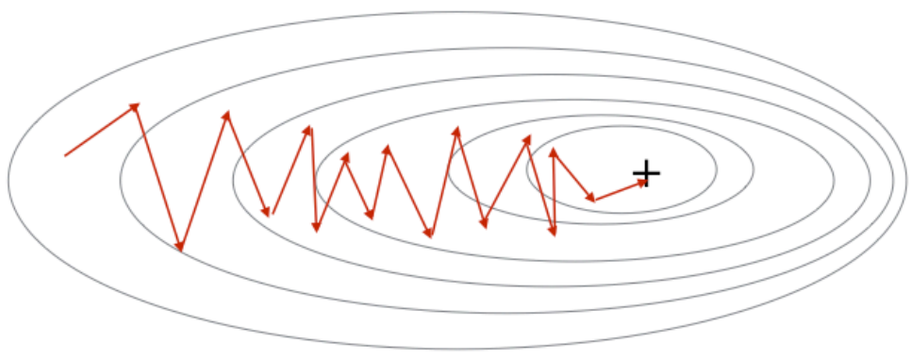
Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Learning rate optimization

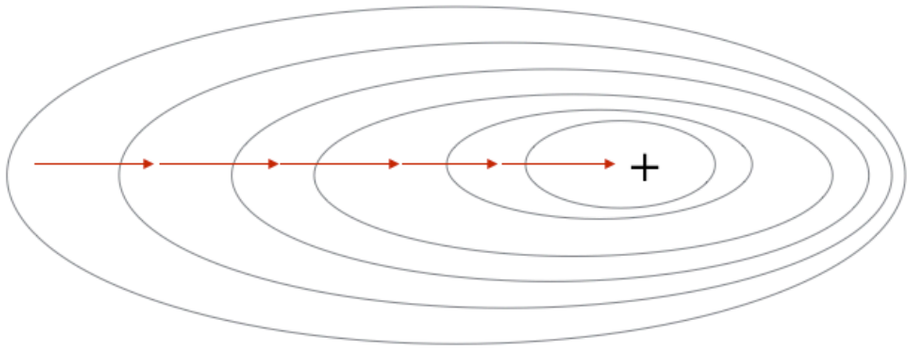
Minibatch learning recap

- Minibatch learning is a form of **stochastic gradient descent**
- Each minibatch can be considered a sample drawn from the training set
- **Noisy gradient**, which can be:
  - ✓ **good**: chance to escape local minima
  - ✓ **bad**: can lead to oscillations
- **Main advantage**: Opportunities to parallelis => **faster convergence**

Stochastic Gradient Descent



Gradient Descent



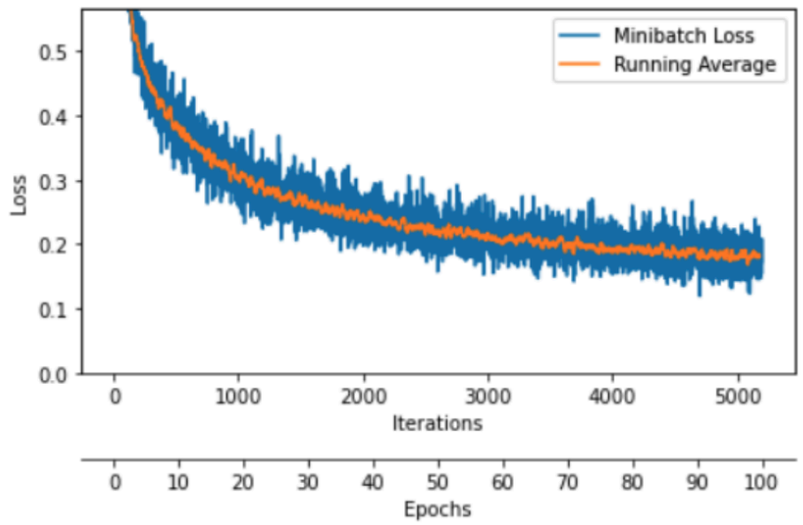
Source: this post

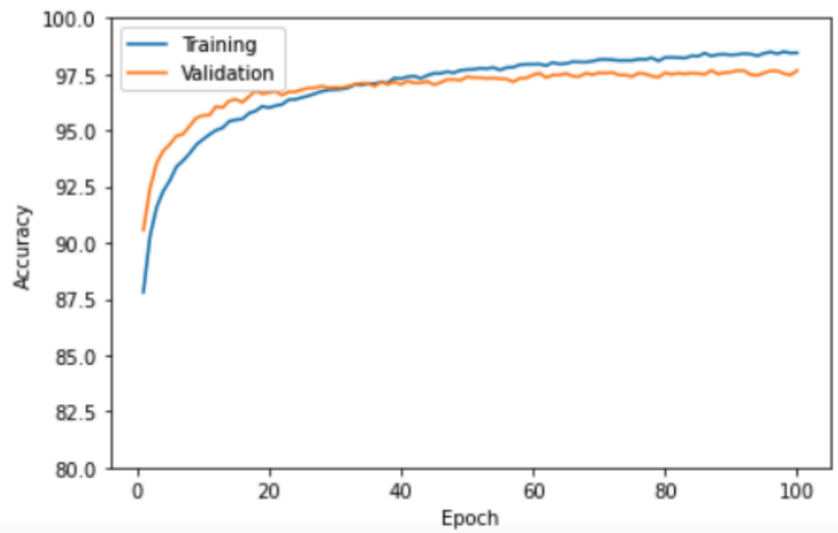
Practical tip for minibatch use

- Reasonable minibatch sizes: 32, 64, 128, 256, …, 1024
- Usually, choose the batch size as large as your GPU allows
- Another practical tip: batch size proportional to number of classes in dataset

```
Epoch: 100/100 | Train: 98.45% | Validation: 97.67%
Time elapsed: 4.38 min
Total Training Time: 4.38 min
Test accuracy 97.08%
```

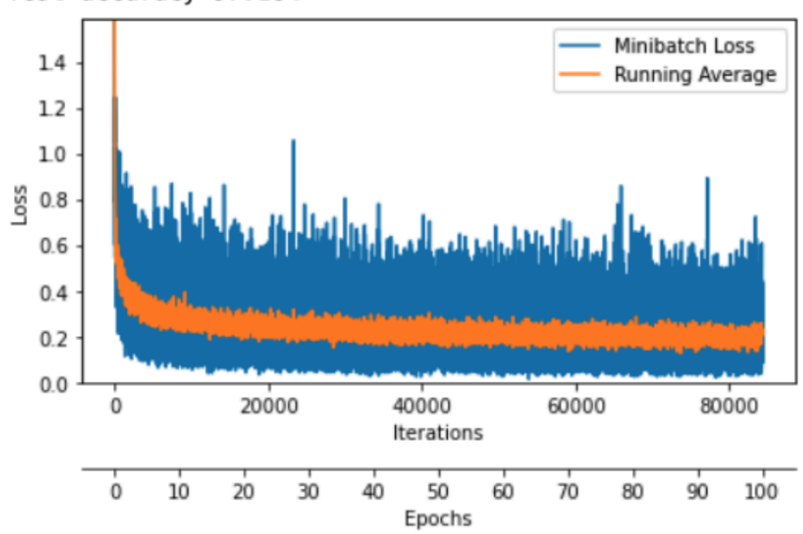
batch size: 1024

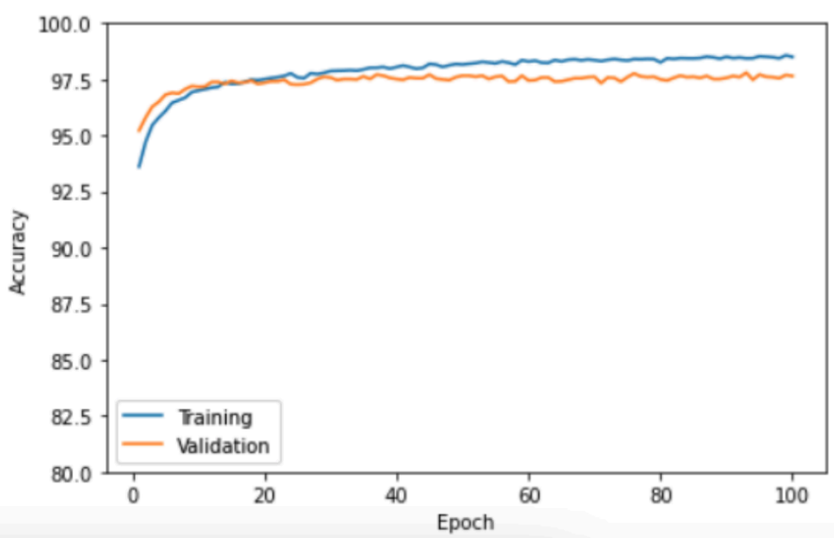





```
Epoch: 100/100 | Train: 98.50% | Validation: 97.65%
Time elapsed: 5.59 min
Total Training Time: 5.59 min
Test accuracy 97.18%
```

batch size: 64





Learning Rate Decay

- Minibatches => loss and gradients are approximations => Oscillations
- To **avoid oscillations at the end of training** => **decay learning rate**
- **Be careful a don't decrease learning rate too early**
  - ✓Train model without learning rate decay first
  - ✓Use validation performance to judge learning rate effect
- Most common variants
  - ✓Exponential decay
  - ✓Halving learning rate
  - ✓Inverse decay
  - ✓**Warm-up**
  - ✓**Cyclic learning rate**
  - ✓**Cosine annealing**

} **Combinations of these are state-of-the-art today**

Common Classical Learning Rate Decay Strategies

1. Exponential rate decay:

$$\eta_t := \eta_0 \cdot e^{-k \cdot t}$$

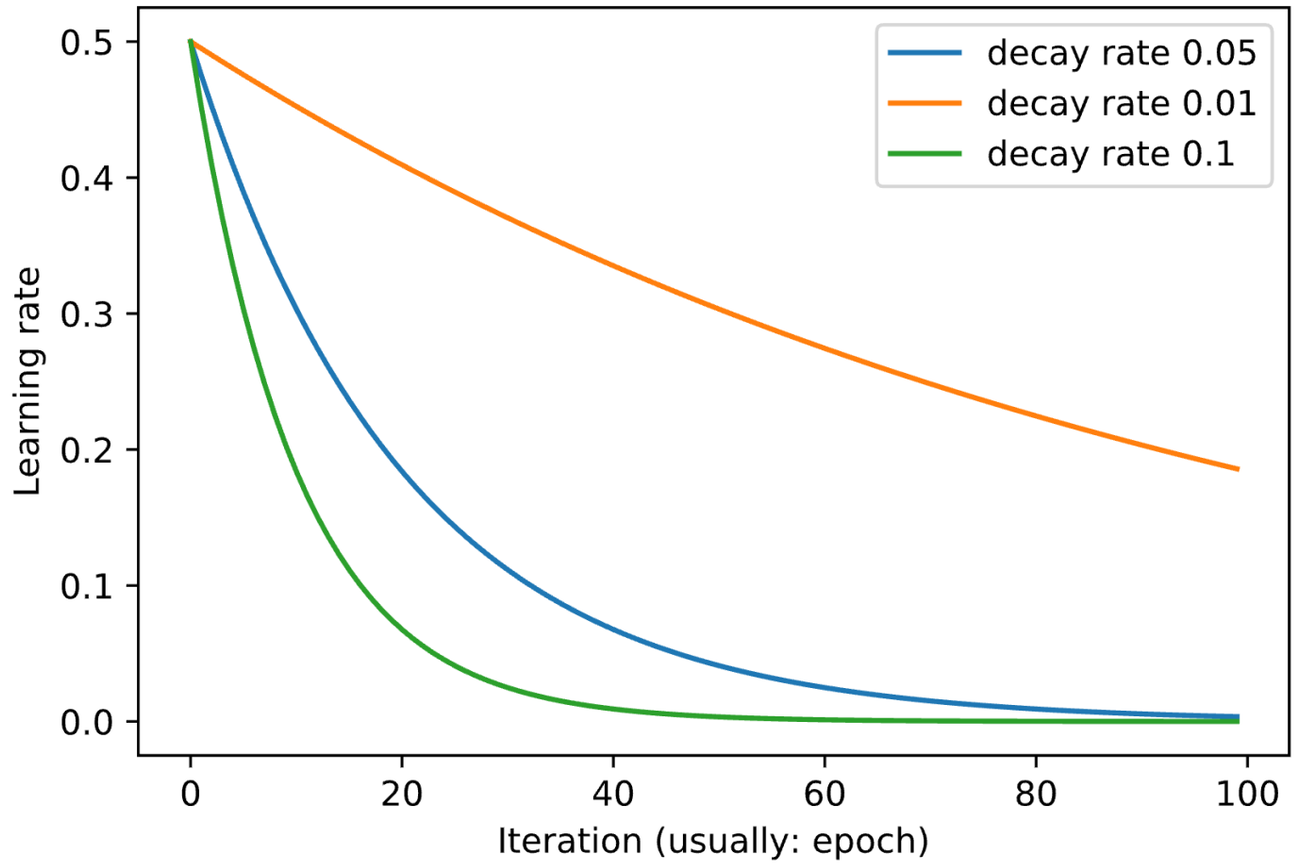
where k is the decay rate

2. Halving the learning rate:

$$\eta_t := \eta_{t-1}/2$$

3. Inverse decay:

$$\eta_t := \frac{\eta_0}{1 + k \cdot t}$$



sebastianraschka.com

Other Variants

1. Cyclic learning rate:

Smith, Leslie N. "Cyclical learning rates for training neural networks." Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on. IEEE, 2017

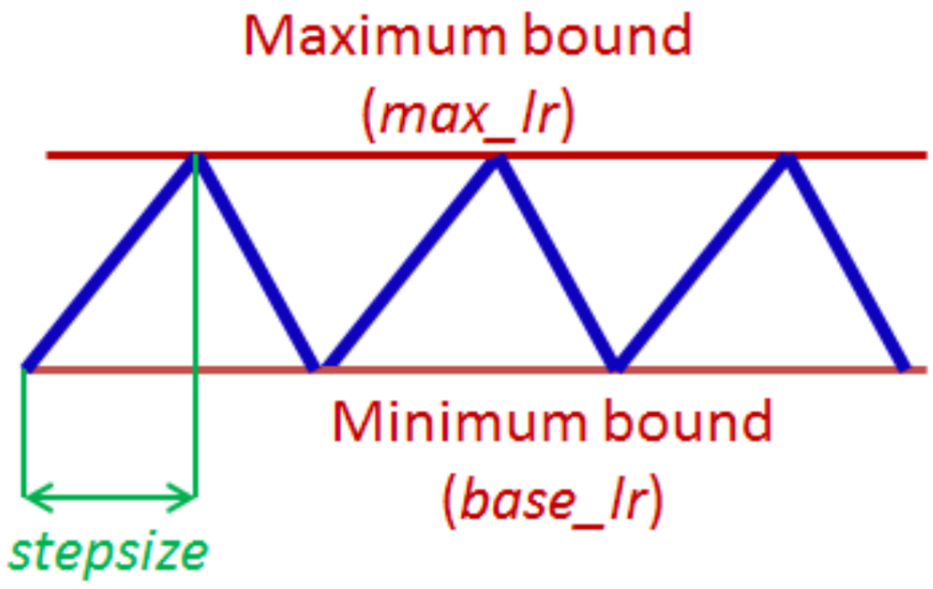


Figure 2. Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter $stepsize$ is the number of iterations in half a cycle.

2. Warm-up

3. Cyclic learning rate

4. Cosine annealing

Relationship Between Learning Rate and Batch Size

DON'T DECAY THE LEARNING RATE, INCREASE THE BATCH SIZE

Samuel L. Smith*, Pieter-Jan Kindermans*, Chris Ying & Quoc V. Le
Google Brain
{slsmith, pikinder, chrisying, qvl}@google.com

ABSTRACT

It is common practice to decay the learning rate. Here we show one can usually obtain the same learning curve on both training and test sets by instead increasing the batch size during training. This procedure is successful for stochastic gradient descent (SGD), SGD with momentum, Nesterov momentum, and Adam. It reaches equivalent test accuracies after the same number of training epochs, but with fewer parameter updates, leading to greater parallelism and shorter training times. We can further reduce the number of parameter updates by increasing the learning rate ϵ and scaling the batch size $B \propto \epsilon$. Finally, one can increase the momentum coefficient m and scale $B \propto 1/(1-m)$, although this tends to slightly reduce the test accuracy. Crucially, our techniques allow us to repurpose existing training schedules for large batch training with no hyper-parameter tuning. We train ResNet-50 on ImageNet to 76.1% validation accuracy in under 30 minutes.

Smith, S. L., Kindermans, P. J., Ying, C., & Le, Q. V. (2017). Don't decay the learning rate, increase the batch size

Relationship Between Learning Rate and Batch Size

Figure 6: Inception-ResNet-V2 on ImageNet. Increasing the batch size during training achieves similar results to decaying the learning rate, but it reduces the number of parameter updates from just over 14000 to below 6000. We run each experiment twice to illustrate the variance.

Smith, S. L., Kindermans, P. J., Ying, C., & Le, Q. V. (2017). Don't decay the learning rate, increase the batch size

Learning Rate in Pytorch

There are built-in functions in Pytorch.

Example of a generic version:

CLASS `torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda, last_epoch=-1)` [SOURCE]

Sets the learning rate of each parameter group to the initial lr times a given function. When last_epoch=-1, sets initial lr as lr.

Parameters:

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **lr_lambda** (*function or list*) – A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups.
- **last_epoch** (*int*) – The index of last epoch. Default: -1.

Example

```
>>> # Assuming optimizer has two groups.
>>> lambda1 = lambda epoch: epoch // 30
>>> lambda2 = lambda epoch: 0.95 ** epoch
>>> scheduler = LambdaLR(optimizer, lr_lambda=[lambda1, lambda2])
>>> for epoch in range(100):
>>>     scheduler.step()
>>>     train(...)
>>>     validate(...)
```

Source: https://pytorch.org/docs/stable/optim.html

Other ways to combat vanishing gradient problem

Residual layers (Skip connections)

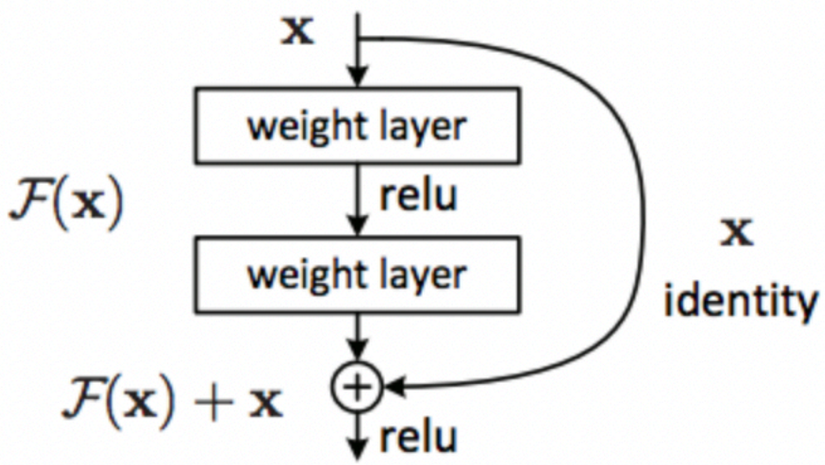


Figure 2. Residual learning: a building block.

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x}.$$

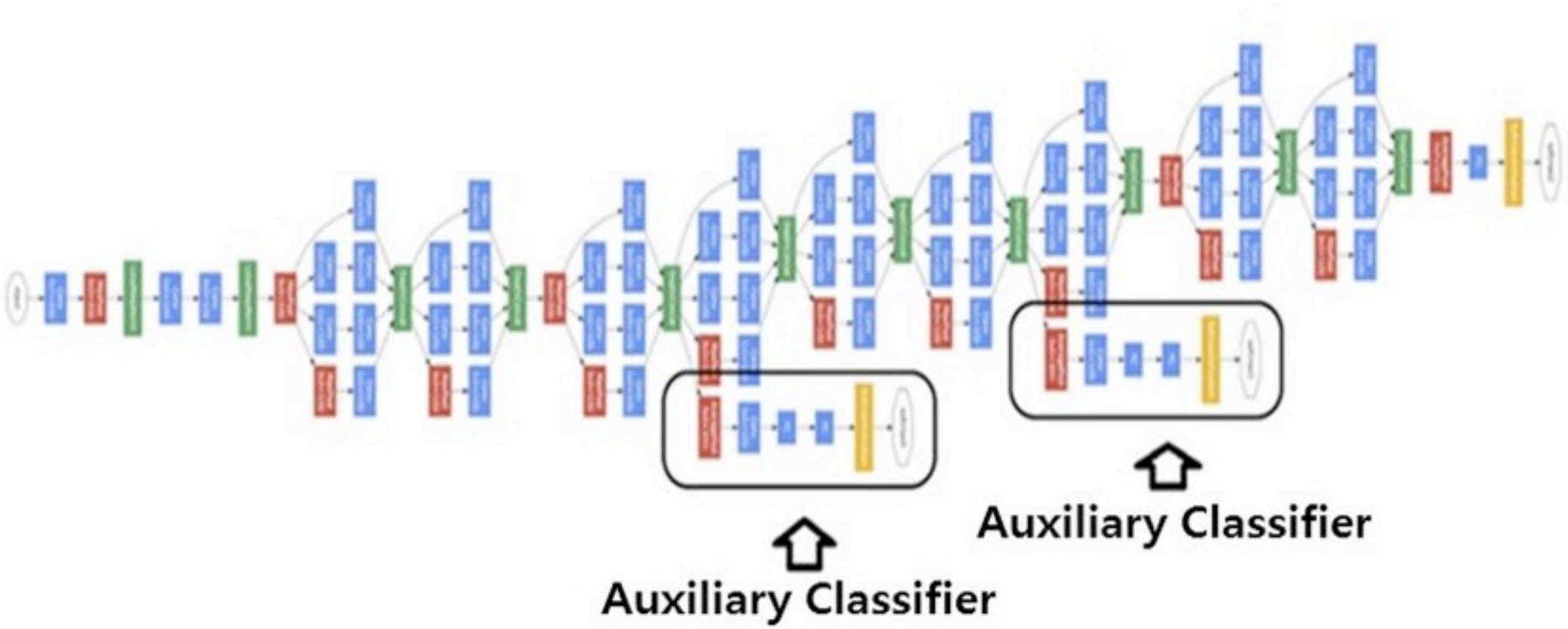
avoid the problem of vanishing gradients,[5] thus leading to easier to optimize neural networks

- Residual networks avoid the vanishing gradient problem by **introducing short paths** which can carry gradient throughout the extent of very deep networks.
- They can be seen as a **collection of many paths of differing length**, they enable very deep networks by leveraging only the short paths during training.
- Layers model increments rather than absolute transformations.

Auxiliary predictors

- Evaluate loss function at multiple points of the network during training, optimize a linear combination of these losses.

- Popularised after Google Inception Network.

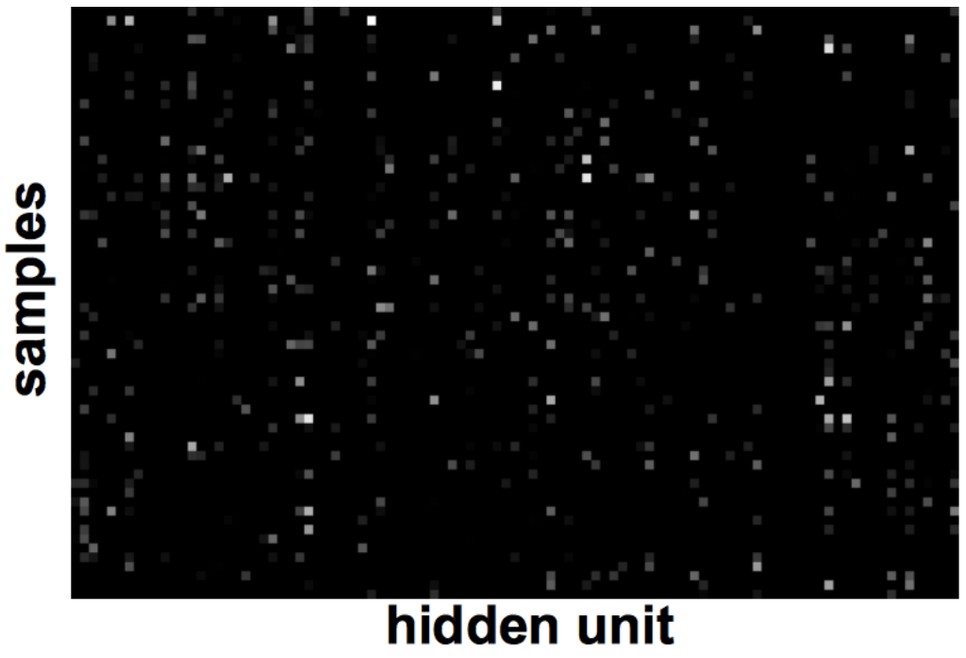


Usual Practice

- Given a dataset $\mathcal{D}_{\text{train}}$, pick a model so that:
  - ✓ You can achieve 0 training error– **Overfit** on the training set
- **Regularize** the model
- SGD with Adam or Momentum, early stopping and Dropout
- Pick learning rate by running on a subset of the data
  - ✓Start with **large learning** rate & divide by 2 until loss does not diverge
- **Decay learning rate** by a factor of ~100 or more by the end of training
- **Modern approach**: **Warm-up** with **cosine annealing** or **OneCycleLR**
- Use **ReLU** (or the updated Leaky ReLU or GELU) activation functions
- Initialize parameters so that each feature across layers has similar variance. **Avoid units in saturation**.

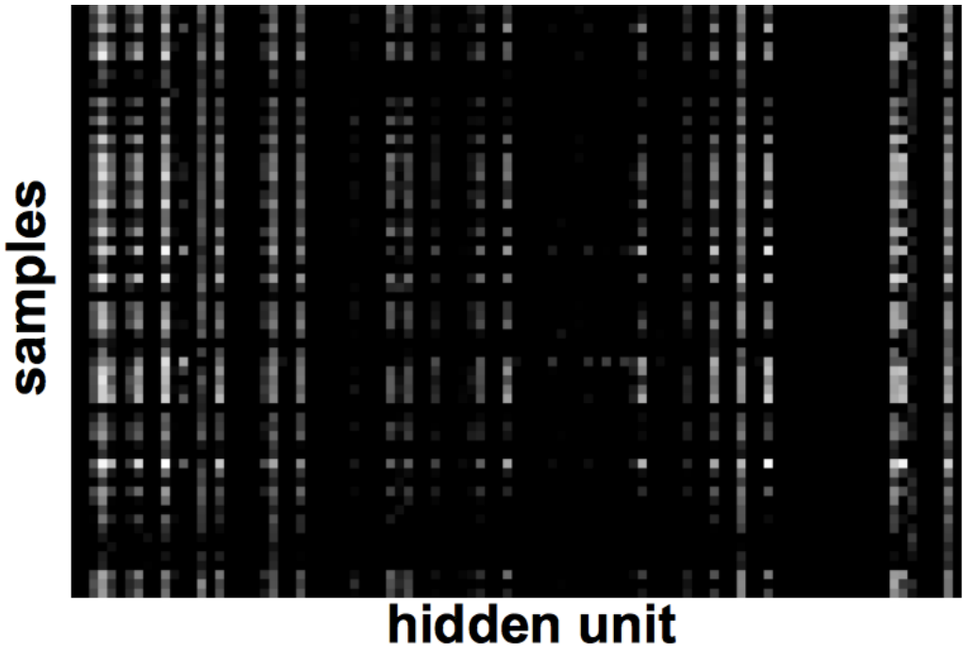
Feature Visualization

Visualize features (features need to be **uncorrelated**) and have **high variance**



Good Training

Hidden units are sparse across samples



BAD Training

Many hidden units ignore the input and/or exhibit strong correlations

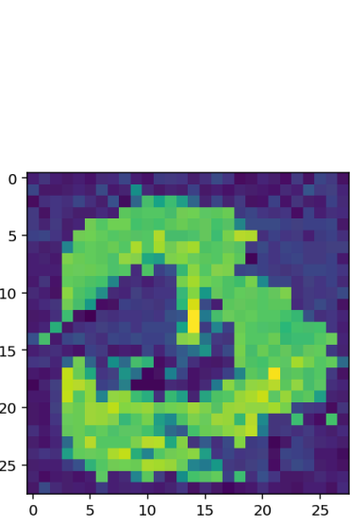
Feature explainability with activation maximization techniques

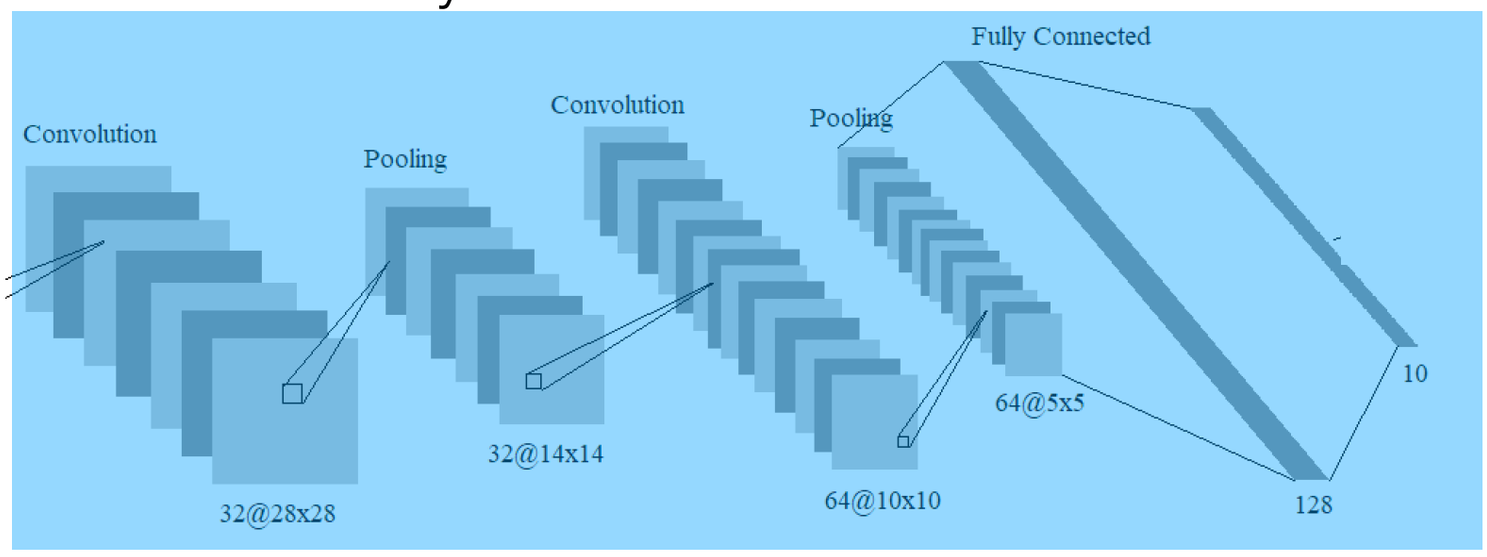
uc3m

Activation maximisation

- Given a trained network, ¿can we identify the most relevant features for a given output?
- With *activation maximization* we fix both the network weights and a certain output confidence level (or an internal representation for a given input).
- We then optimise by GD a random input to achieve such an output
- In practice only some input features are required.

My favourite Neural Net here





$$y' = \begin{bmatrix} 0.01 \\ 0.08 \\ 0.09 \\ 0.12 \\ 0.13 \\ 0.17 \\ 0.15 \\ 0.05 \\ 0.14 \\ 0.06 \end{bmatrix}$$

$$\frac{\partial ||y - y'||^2}{\partial x}$$

Grad. Descent

Backpropagation formula: proof

Computing Gradients: Backpropagation

Assume an M-layer MLP

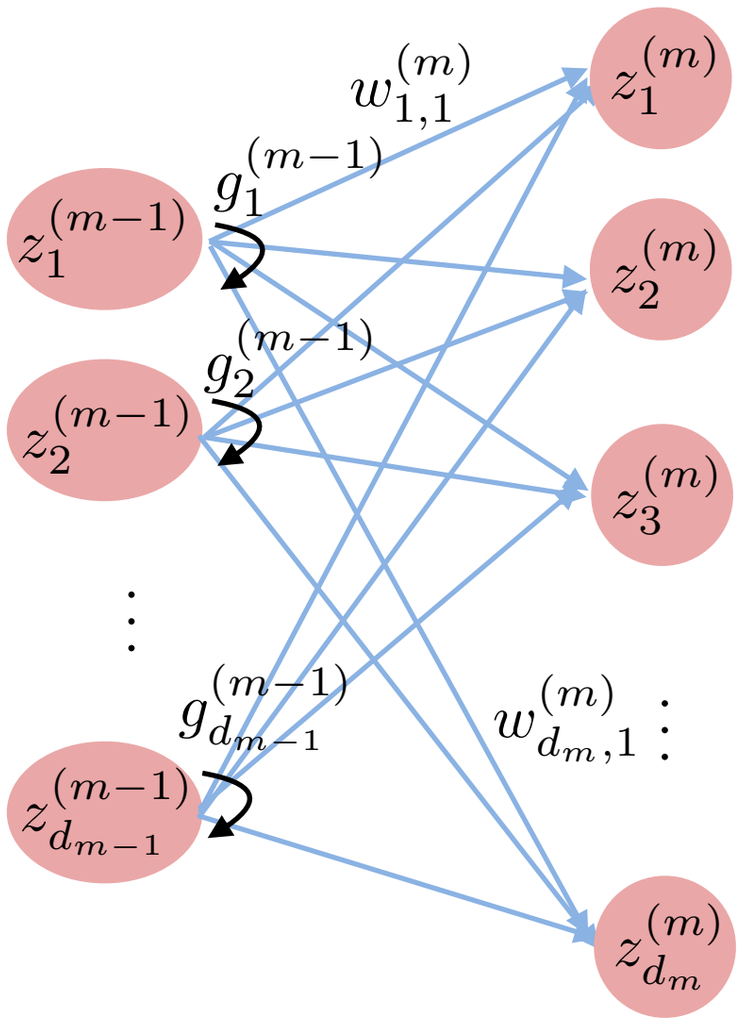
$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} \longrightarrow \mathbf{g}^{(1)} = g(\mathbf{z}^{(1)}) \longrightarrow \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{g}^{(1)} \longrightarrow \cdots \longrightarrow \hat{\mathbf{y}} = g(\mathbf{z}^{(M)})$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) = \sum_{n=1}^{N} \mathcal{L}_n$$

$$\frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m)}} \frac{\partial z_j^{(m)}}{\partial w_{ji}^{(m)}} \longrightarrow \delta_j^{(m)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m)}}$$

$$\frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}} = \delta_j^{(m)} g_i^{(m-1)}$$

$$z_j^{(m)} = \sum_{i=1}^{d_{m-1}} w_{j,i}\ g_i^{(m-1)} \longrightarrow \frac{\partial z_j^{(m)}}{\partial w_{ji}^{(m)}} = g_i^{(m-1)}$$



Computing Gradients: Error Backpropagation

$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} \longrightarrow \mathbf{g}^{(1)} = g(\mathbf{z}^{(1)}) \longrightarrow \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{g}^{(1)} \longrightarrow \cdots \longrightarrow \hat{\mathbf{y}} = g(\mathbf{z}^{(M)})$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) = \sum_{n=1}^{N} \mathcal{L}_n \qquad \frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}} = \delta_j^{(m)} g_i^{(m-1)} \qquad \delta_j^{(m)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m)}}$$

Linear output activation

$$\hat{\mathbf{y}} = \mathbf{z}^{(M)} \qquad \mathcal{L}_n = ||\mathbf{y} - \hat{\mathbf{y}}||^2$$

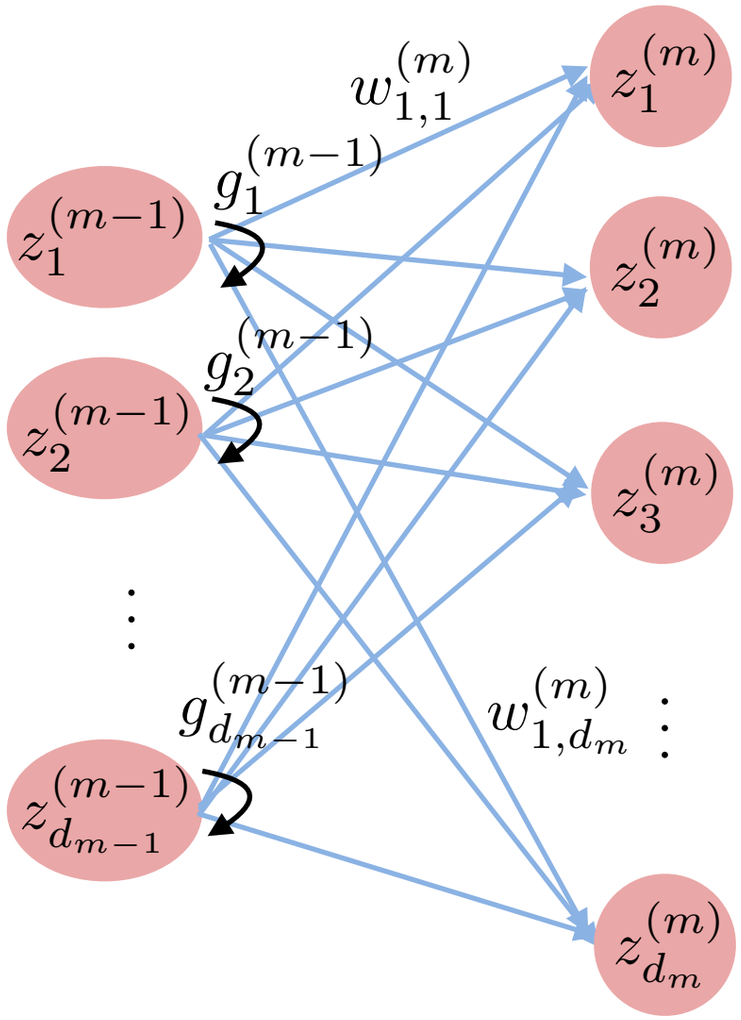
$$\delta_j^{(M)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(M)}} = \frac{\partial \mathcal{L}_n}{\partial \hat{y}_j} = -2(y_j - \hat{y}_j)$$

Sigmoid output activation

$$\hat{y}_j = \frac{1}{1 + \exp(-z_j^{(M)})}$$

$$\mathcal{L}_n = \sum_{i=1}^{d} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

$$\delta_j^{(M)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(M)}} = -(y_j - \hat{y}_j)$$



Computing Gradients: Error Backpropagation

$$\mathbf{x} \longrightarrow \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} \longrightarrow \mathbf{g}^{(1)} = g(\mathbf{z}^{(1)}) \longrightarrow \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{g}^{(1)} \longrightarrow \cdots \longrightarrow \hat{\mathbf{y}} = \mathbf{W}^{(M)}\mathbf{g}^{(M-1)}$$

$$J(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}\left(\hat{\mathbf{y}}^{(n)}, \mathbf{y}^{(n)}\right) = \sum_{n=1}^{N} \mathcal{L}_n \qquad \frac{\partial \mathcal{L}_n}{\partial w_{ji}^{(m)}} = \delta_j^{(m)} g_i^{(m-1)} \qquad \delta_j^{(m)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m)}}$$

$$\delta_j^{(m-1)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(m-1)}} = \sum_{k=1}^{d_m} \frac{\partial \mathcal{L}_n}{\partial z_k^{(m)}} \frac{\partial z_k^{(m)}}{\partial z_j^{(m-1)}}$$

$$\frac{\partial z_k^{(m)}}{\partial z_j^{(m-1)}} = w_{k,j}\ g'(z_j^{(m-1)})$$

Learning representations
by back-propagating errors

Rumelhart, Hinton, Williams (1986)

$$\delta_j^{(m-1)} = g'(z_j^{(m-1)}) \sum_{k=1}^{d_m} w_{k,j}\delta_k^{(m)}$$