Normalisation and Initialisation

Steve Renals

Machine Learning Practical — MLP Lecture 6
26 October 2016

Feature Normalisation

Data Preprocessing

- Normalisation
  - Subtract the mean of the input data from every feature, and scale by its standard deviation

$$\hat{x_i^n} = \frac{x_i^n - \text{mean}(x_i)}{\text{sd}(x_i)}$$

- PCA - Principal Components Analysis
  - Decorrelate the data by projecting onto the principal components
  - Also possible to reduce dimensionality by only projecting onto the top P principal components
- Whitening
  - Decorrelate by PCA
  - Scale each dimension

PCA and Whitening

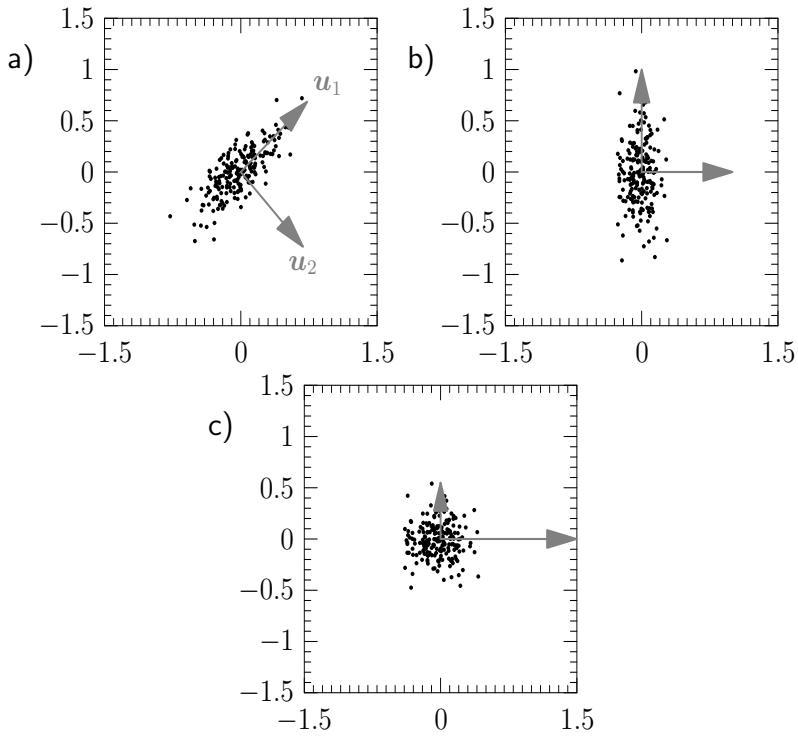


Figure 3: Illustration of PCA and whitening. a) The original data "cloud". The arrows show the principal components. The first one points in the direction of the largest variance in the data, and the second in the remaining orthogonal direction. b) When the data is transformed to the principal components, i.e. the principal components are taken as the new coordinates, the variation in the data is aligned with those new axes, which is because the principal components are uncorrelated. c) When the principal components are further normalized to unit variance, the data cloud has equal variance in all directions, which means it has been whitened. The change in the lengths of the arrows reflects this normalization; the larger the variance, the shorter the arrow.

from Hyvärinen et al, *Natural Image Statistics*, Springer, 2009.

How should we initialise deep networks?

Random weight initialisation

- Initialise weights to small random numbers r, sampling weights independently from a Gaussian or from a uniform distribution
  - control the initialisation by setting the mean (typically to 0) and variance of the weight distribution
- Biases may be initialised to 0
  - output (softmax) biases can be normalised to $\log(p(c))$, log of prior probability of the corresponding class c
- Calibration – variance of the input to a unit independent of the number of incoming connections ("fan-in", n_{in})
- Heuristic: $w_i \sim U(-\sqrt{1/n_{in}}, \sqrt{1/n_{in}})$ [U is uniform distribution]
  - Corresponds to a variance $\mathrm{Var}(w_i) = 1/(3n_{in})$
  - (Since, if $x \sim U(a, b)$, then $\mathrm{Var}(x) = (b-a)^2/12$ so if $x \sim U(-n, n)$, then $\mathrm{Var}(x) = n^2/3$)

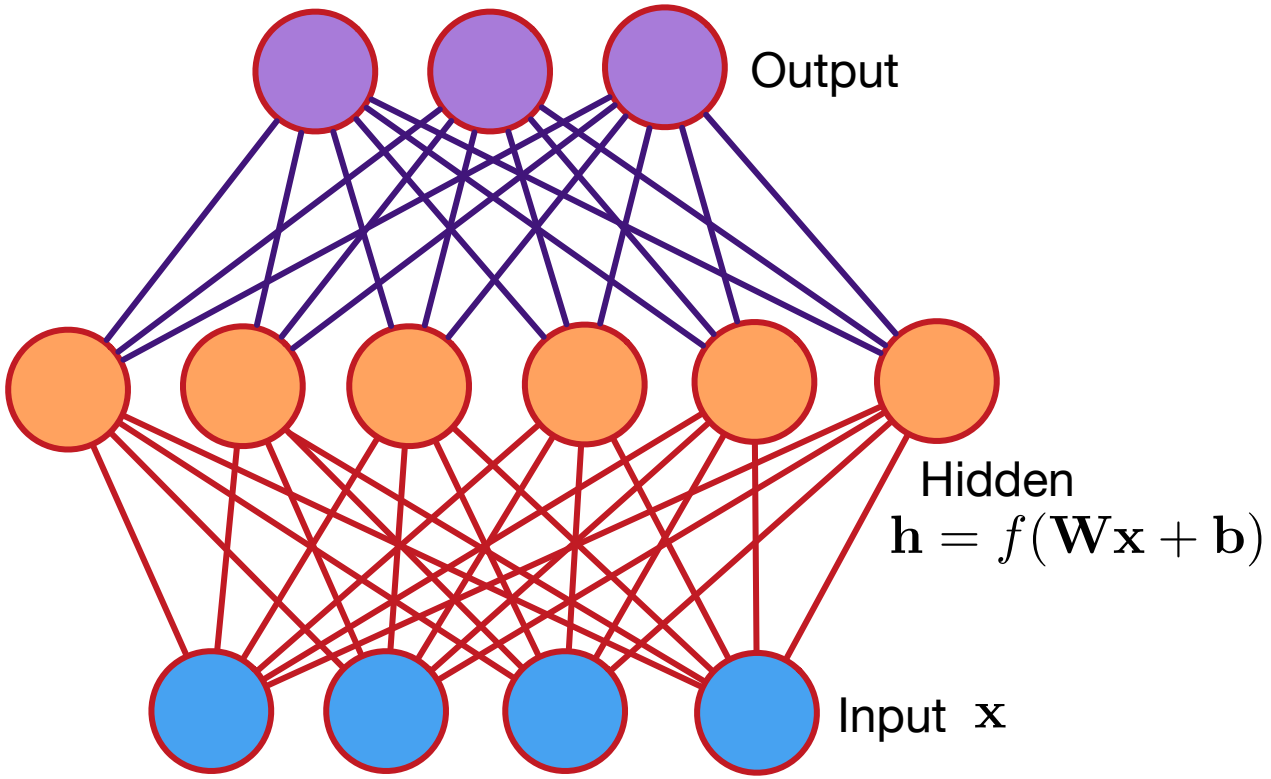
"GlorotInit" ("Xavier initialisation")

- We would like to constrain the variance of each layer to be $1/n_{in}$, thus
$$w_i \sim U(-\sqrt{3/n_{in}}, \sqrt{3/n_{in}})$$
- However we need to take the backprop into account, hence we would also like $\text{Var}(w_i) = 1/n_{out}$
- As compromise set the variance to be $\text{Var}(w_i) = 2/(n_{in} + n_{out})$
- This corresponds to Glorot and Bengio's normalised initialisation
$$w_i \sim U\left(-\sqrt{6/(n_{in} + n_{out})}, \sqrt{6/(n_{in} + n_{out})}\right)$$

Glorot and Bengio, "Understanding the difficulty of training deep feedforward networks", *AISTATS*, 2010.

http://www.jmlr.org/proceedings/papers/v9/glorot10a.html

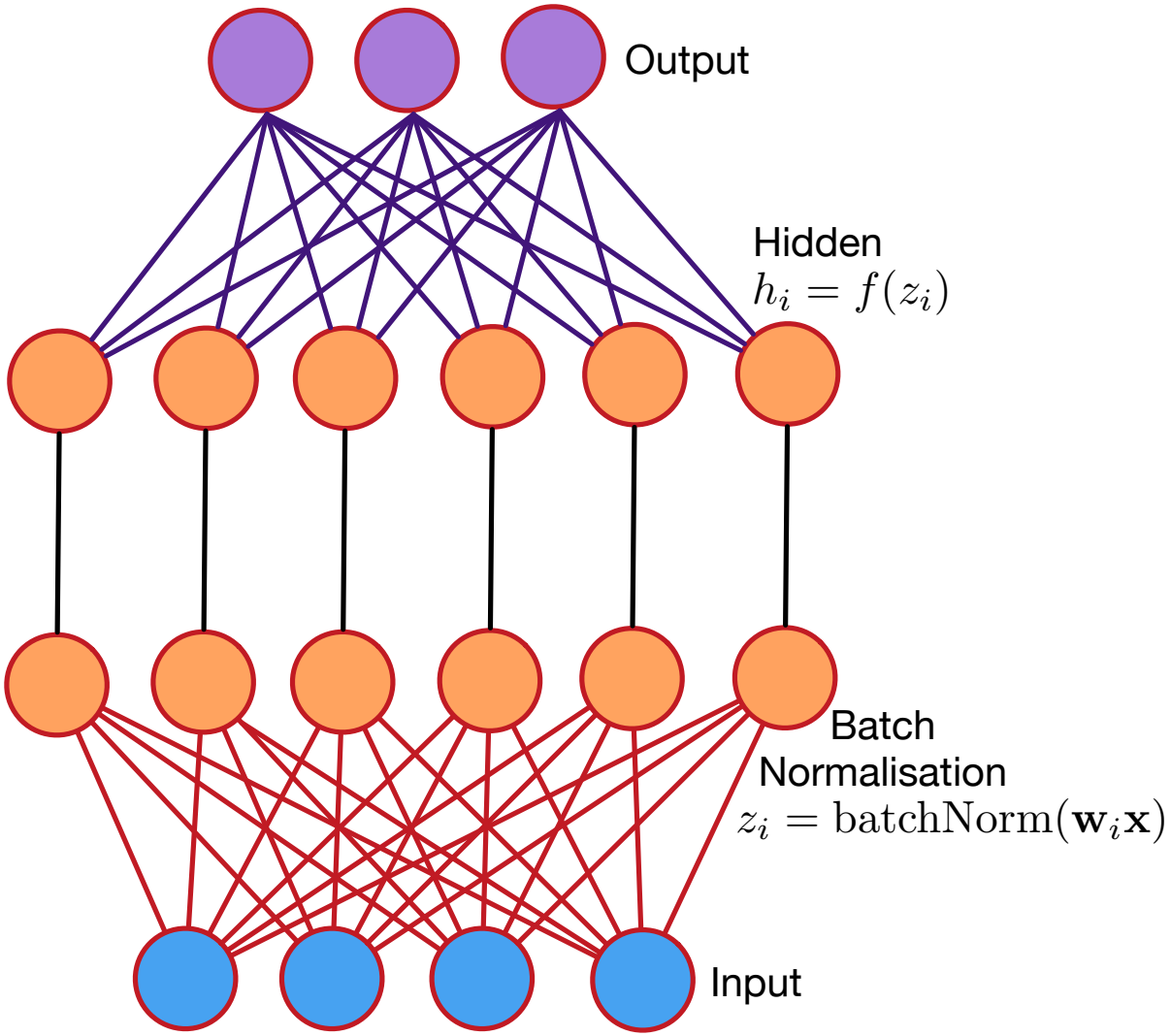
Batch Normalisation



Ioffe & Szegedy, "Batch normalization", ICML-2015

http://www.jmlr.org/proceedings/papers/v37/ioffe15.html

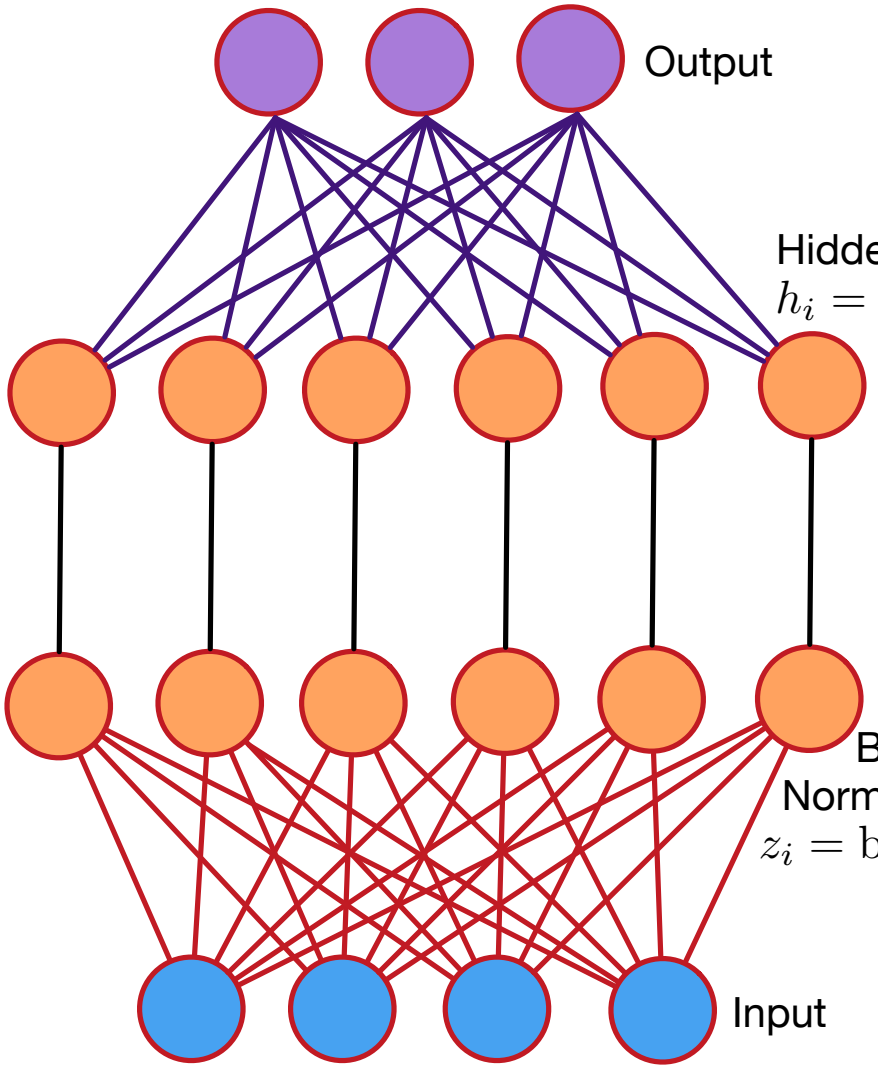
Batch Normalisation



Ioffe & Szegedy, "Batch normalization", ICML-2015

http://www.jmlr.org/proceedings/papers/v37/ioffe15.html

Batch Normalisation



Normalise hidden unit activations , then scale and shift

$$u_i = \mathbf{w}_i\mathbf{x}$$

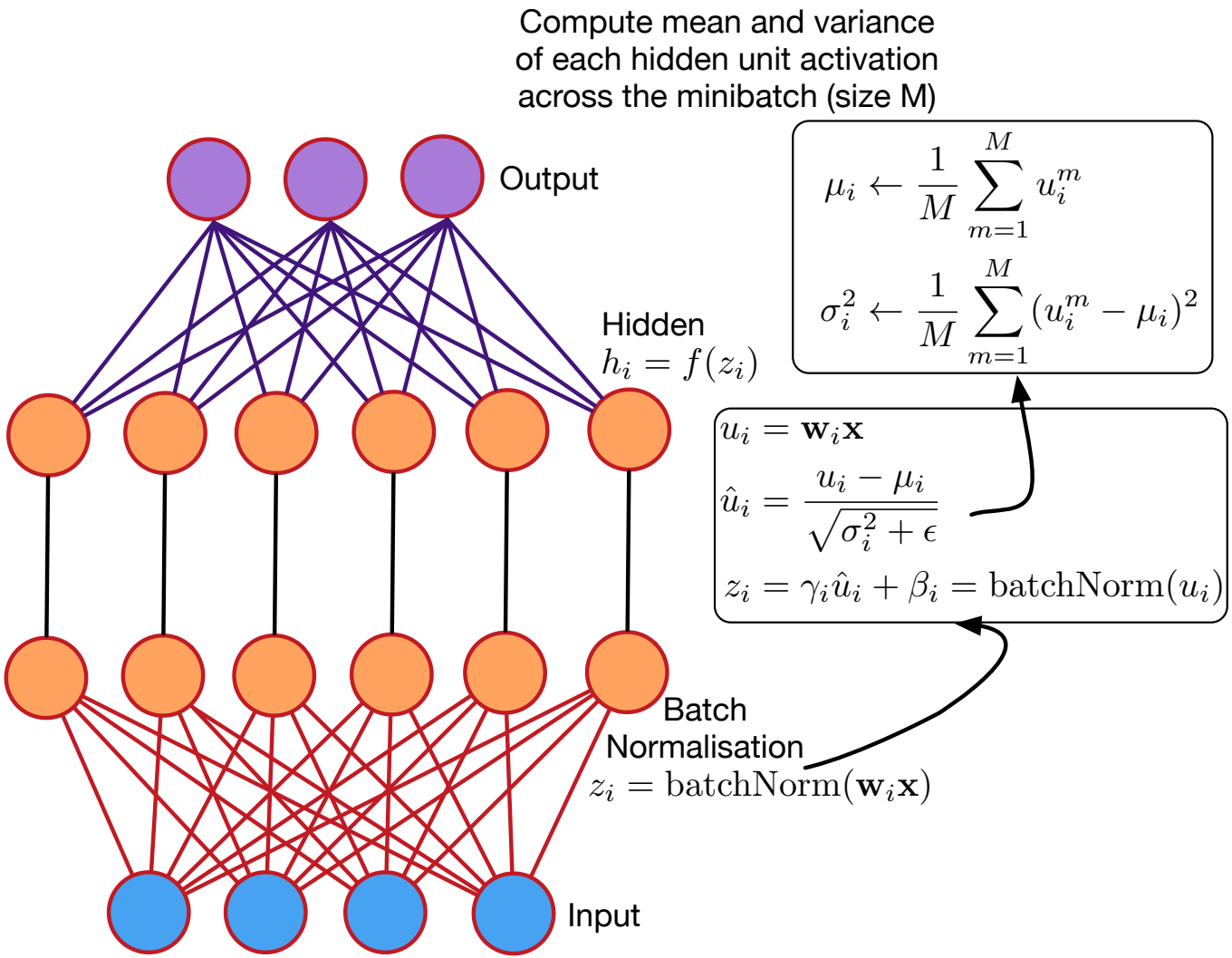
$$\hat{u}_i = \frac{u_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

$$z_i = \gamma_i \hat{u}_i + \beta_i = \text{batchNorm}(u_i)$$

Ioffe & Szegedy, "Batch normalization", ICML-2015

http://www.jmlr.org/proceedings/papers/v37/ioffe15.html

Batch Normalisation



Ioffe & Szegedy, "Batch normalization", ICML-2015

http://www.jmlr.org/proceedings/papers/v37/ioffe15.html

Batch normalisation

- Use minibatch statistics to normalise activations of each layer (activations are the argument of the transfer function)
- Parameters γ and β can scale and shift the normalised activations; β can also play the role of bias
- batchNorm depends on the current training example – and on examples in the minibatch (to compute mean and variance)
- Training
  - Set parameters γ and β by gradient descent – require gradients $\frac{\partial E}{\partial \gamma}$ and $\frac{\partial E}{\partial \beta}$
  - To back-propagate gradients through the batchNorm layer also require: $\frac{\partial E}{\partial \hat{u}}$ $\frac{\partial E}{\partial \sigma^2}$ $\frac{\partial E}{\partial \mu}$ $\frac{\partial E}{\partial u_i}$
- Runtime - use the sample mean and variance computed over the complete training data as the mean and variance parameters for each layer – fixed transform:

$$\hat{u}_i = \frac{u_i - \text{mean}(u_i)}{\sqrt{\text{Var}(u_i) + \epsilon}}$$

Batch normalisation – gradients (for reference)

$$\frac{\partial E}{\partial \hat{u}_i^m} = \frac{\partial E^m}{\partial z_i^m} \cdot \gamma_i$$

$$\frac{\partial E}{\partial \sigma_i^2} = \sum_m \frac{\partial E^m}{\partial \hat{u}_i^m} \cdot (u_i^m - \mu_i) \cdot \frac{-1}{2} \left(\sigma_i^2 + \epsilon\right)^{-3/2}$$

$$\frac{\partial E}{\partial \mu_i} = \left(\sum_m \frac{\partial E^m}{\partial \hat{u}_i^m} \cdot \frac{-1}{\sqrt{\sigma_i^2 + \epsilon}} \right) + \frac{\partial E}{\partial \sigma_i^2} \cdot \frac{1}{M} \sum_m -2(u_i - \mu_i)$$

$$\frac{\partial E}{\partial u_i^m} = \frac{\partial E^m}{\partial \hat{u}_i^m} \cdot \frac{1}{\sqrt{\sigma_i^2 + \epsilon}} + \frac{\partial E}{\partial \sigma_i^2} \cdot \frac{2(u_i - \mu_i)}{M} + \frac{\partial E}{\partial \mu_i} \cdot \frac{1}{M}$$

$$\frac{\partial E}{\partial \gamma_i} = \sum_m \frac{\partial E^m}{\partial z_i^m} \cdot \hat{u}_i^m$$

$$\frac{\partial E}{\partial \beta_i} = \sum_m \frac{\partial E^m}{\partial z_i^m}$$

see also http://cthorey.github.io/backpropagation/

Benefits of batch normalisation

- Makes training many-layered networks easier
  - Allows higher learning rates
  - Weight initialisation less crucial
- Can act like a regulariser – maybe reduces need for techniques like dropout
- Can be applied to convolutional networks
- In practice (image processing) – achieves similar accuracy with many fewer training cycles
- Very widely used, and very useful for many-layered networks (e.g. visual object recognition)

Pretraining

Pretraining

- **Why is training deep networks hard?**
  - Vanishing (or exploding) gradients – gradients for layers closer to the input layer are computed multiplicatively using backprop
  - If sigmoid/tanh hidden units near the output saturate then back-propagated gradients will be very small
  - Good discussion in chapter 5 of *Neural Networks and Deep Learning*

Pretraining

- **Why is training deep networks hard?**
  - Vanishing (or exploding) gradients – gradients for layers closer to the input layer are computed multiplicatively using backprop
  - If sigmoid/tanh hidden units near the output saturate then back-propagated gradients will be very small
  - Good discussion in chapter 5 of *Neural Networks and Deep Learning*
- **Solve by stacked pretraining**
  - Train the first hidden layer
  - Add a new hidden layer, and train only the parameters relating to the new hidden layer. Repeat.
  - The use the pretrained weights to initialise the network – emphfine-tune the complete network using gradient descent

Pretraining

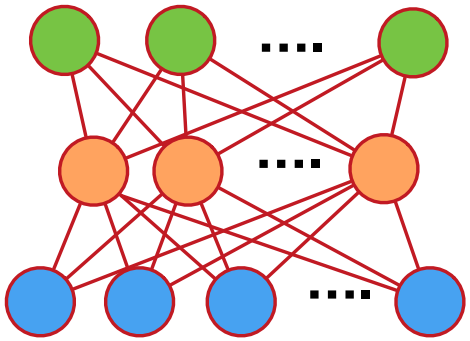
- **Why is training deep networks hard?**
  - Vanishing (or exploding) gradients – gradients for layers closer to the input layer are computed multiplicatively using backprop
  - If sigmoid/tanh hidden units near the output saturate then back-propagated gradients will be very small
  - Good discussion in chapter 5 of *Neural Networks and Deep Learning*
- **Solve by stacked pretraining**
  - Train the first hidden layer
  - Add a new hidden layer, and train only the parameters relating to the new hidden layer. Repeat.
  - The use the pretrained weights to initialise the network – emphfine-tune the complete network using gradient descent
- **Approaches to pre-training**
  - Supervised: Layer-by-layer cross-entropy training
  - Unsupervised: Autoencoders
  - Unsupervised: Restricted Boltzmann machines (not covered in this course)

Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough

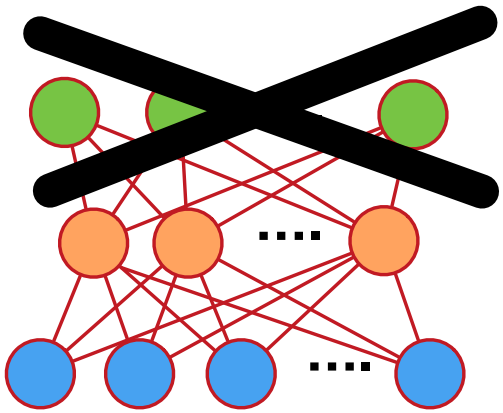
Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough



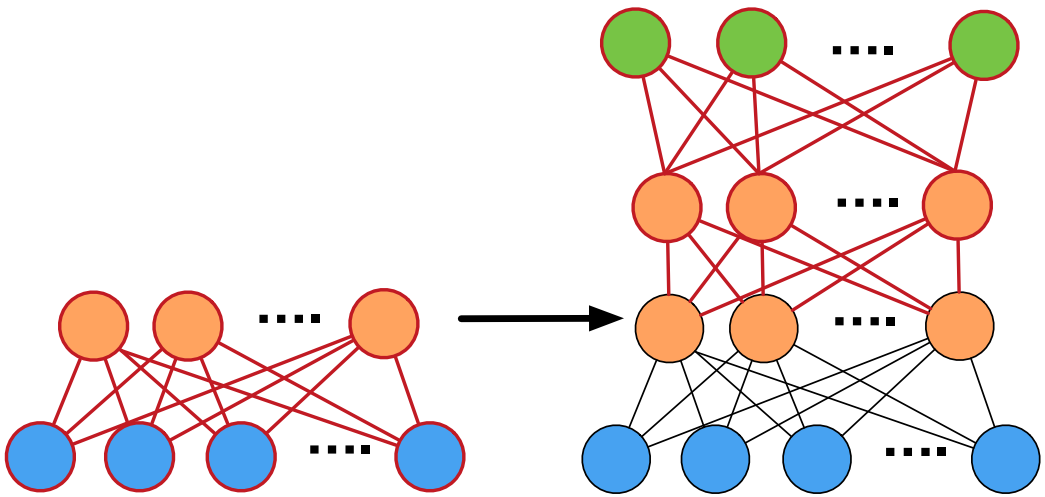
Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough



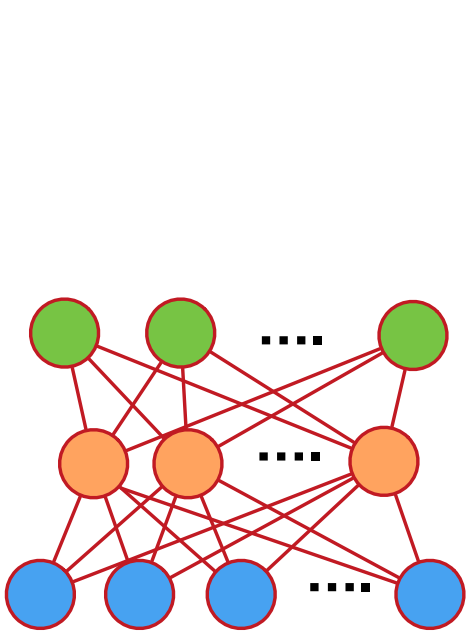
Greedy Layer-by-layer cross-entropy training

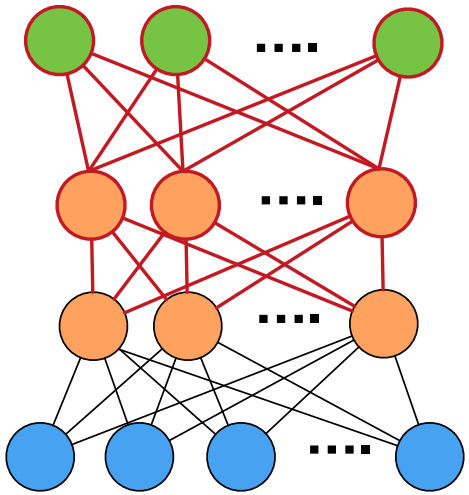
1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough

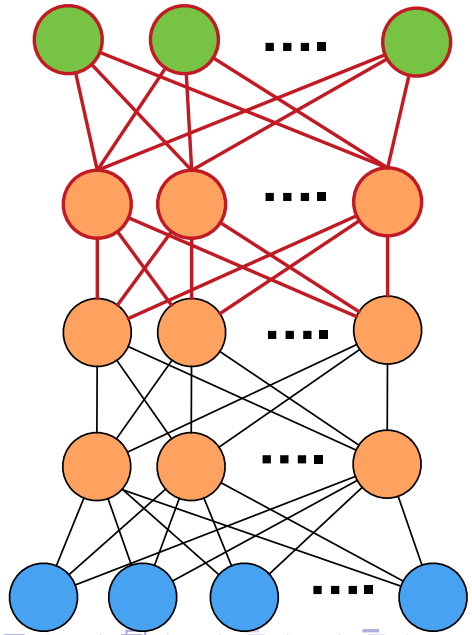


Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough

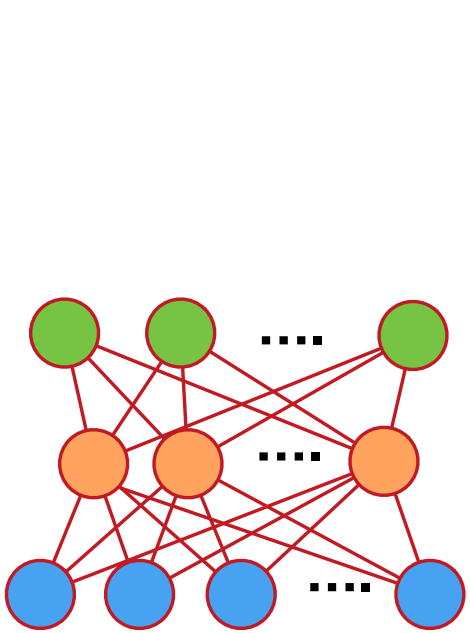


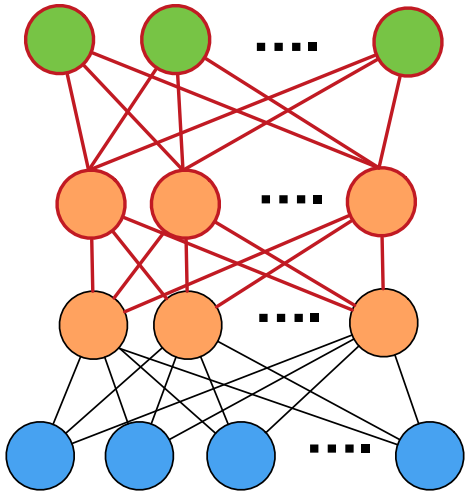


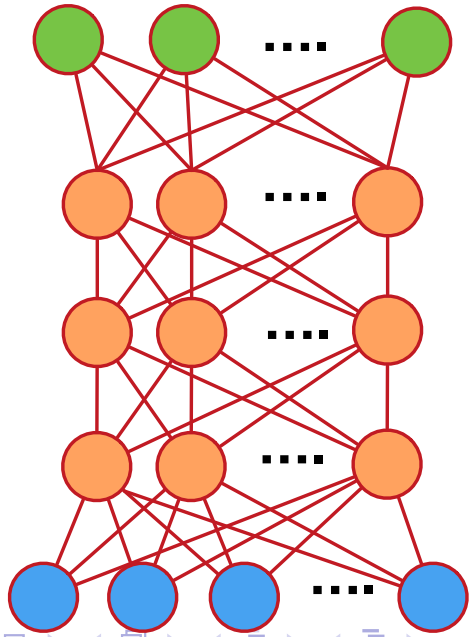


Greedy Layer-by-layer cross-entropy training

1. Train a network with one hidden layer
2. Remove the output layer and weights leading to the output layer
3. Add an additional hidden layer and train only the newly added weights
4. Goto 2 or finetune & stop if deep enough

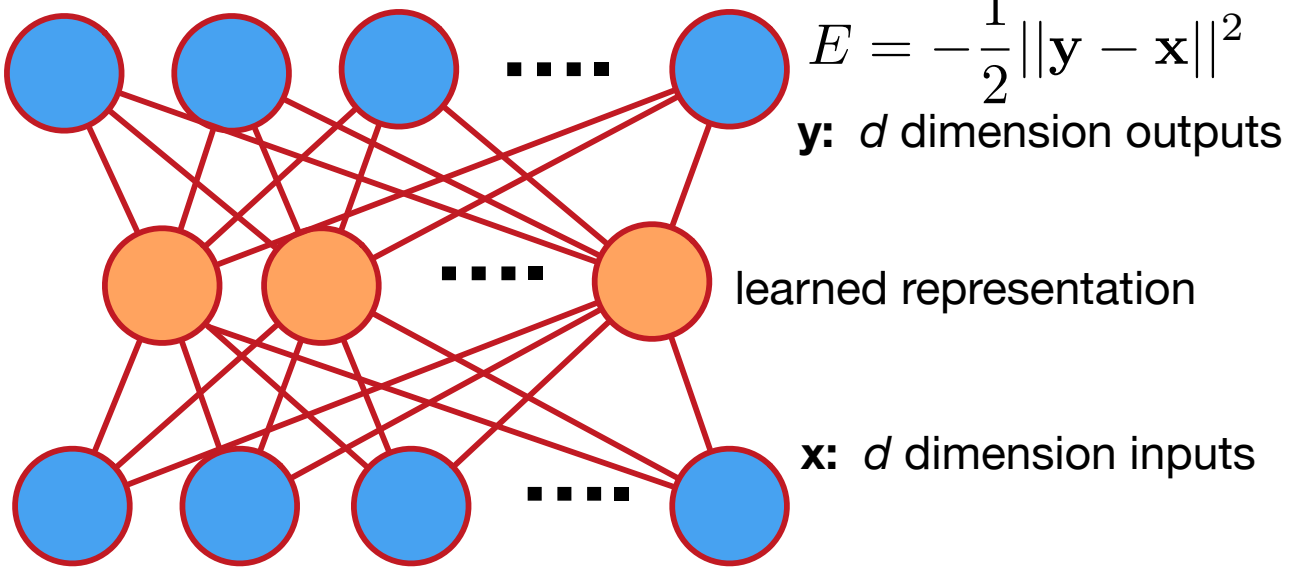






Autoencoders

- An autoencoder is a neural network trained to map its input into a distributed representation from which the input can be reconstructed
- Example: single hidden layer network, with an output the same dimension as the input, trained to reproduce the input using squared error cost function



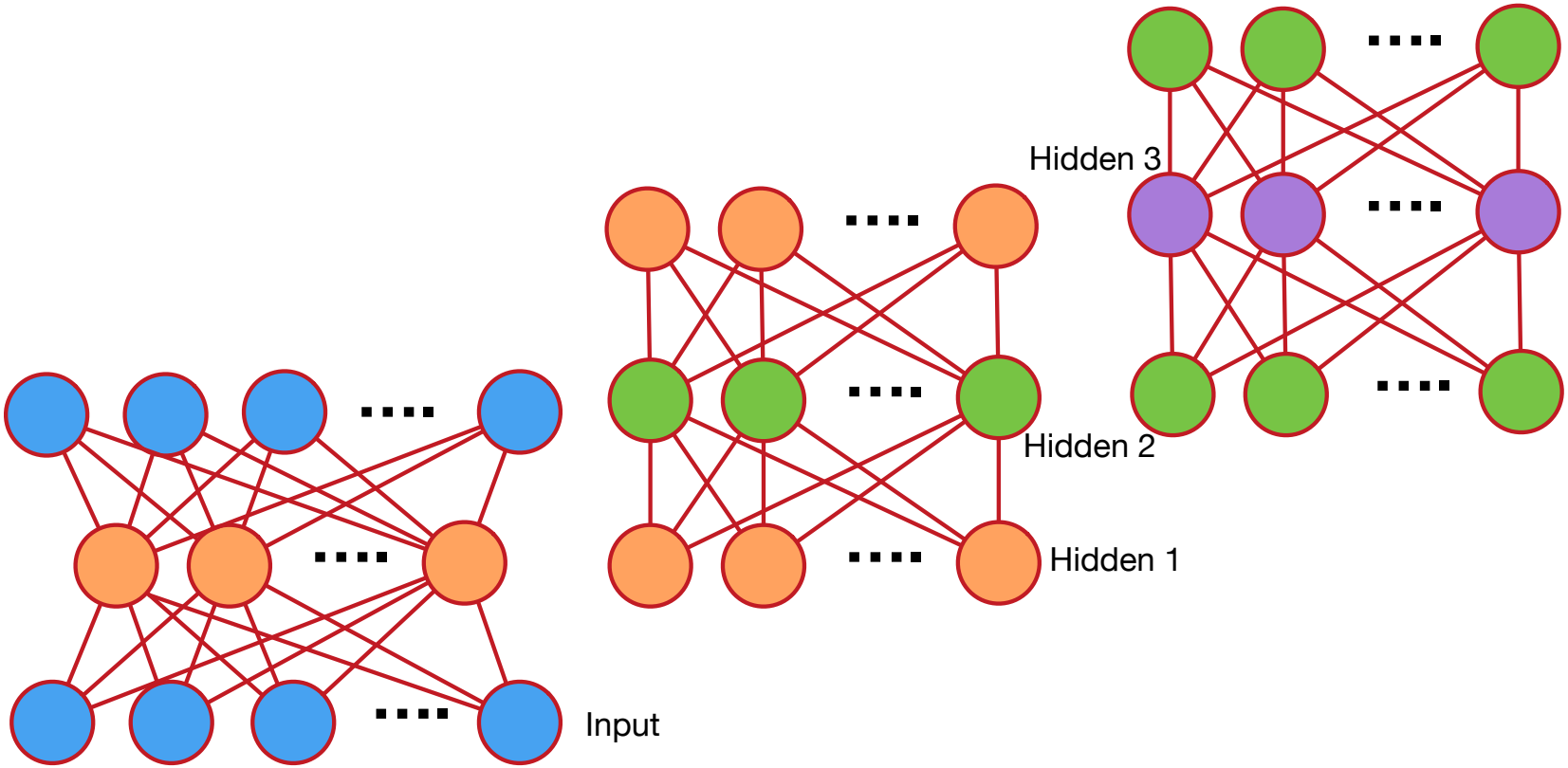
Stacked autoencoders

- Can the hidden layer just copy the input (if it has an equal or higher dimension)?
  - In practice experiments show that nonlinear autoencoders trained with stochastic gradient descent result in useful hidden representations
  - Early stopping acts as a regulariser

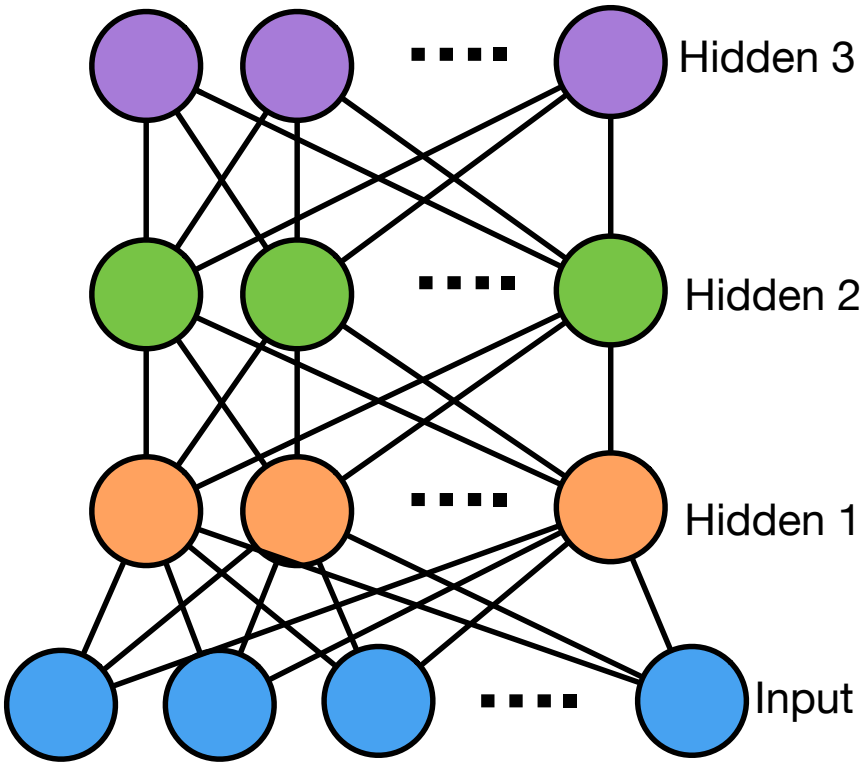
Stacked autoencoders

- Can the hidden layer just copy the input (if it has an equal or higher dimension)?
  - In practice experiments show that nonlinear autoencoders trained with stochastic gradient descent result in useful hidden representations
  - Early stopping acts as a regulariser
- **Stacked autoencoders** – train a sequence of autoencoders, layer-by-layer
  - First train a single hidden layer autoencoder
  - Then use the learned hidden layer as the input to a new autoencoder

Stacked Autoencoders

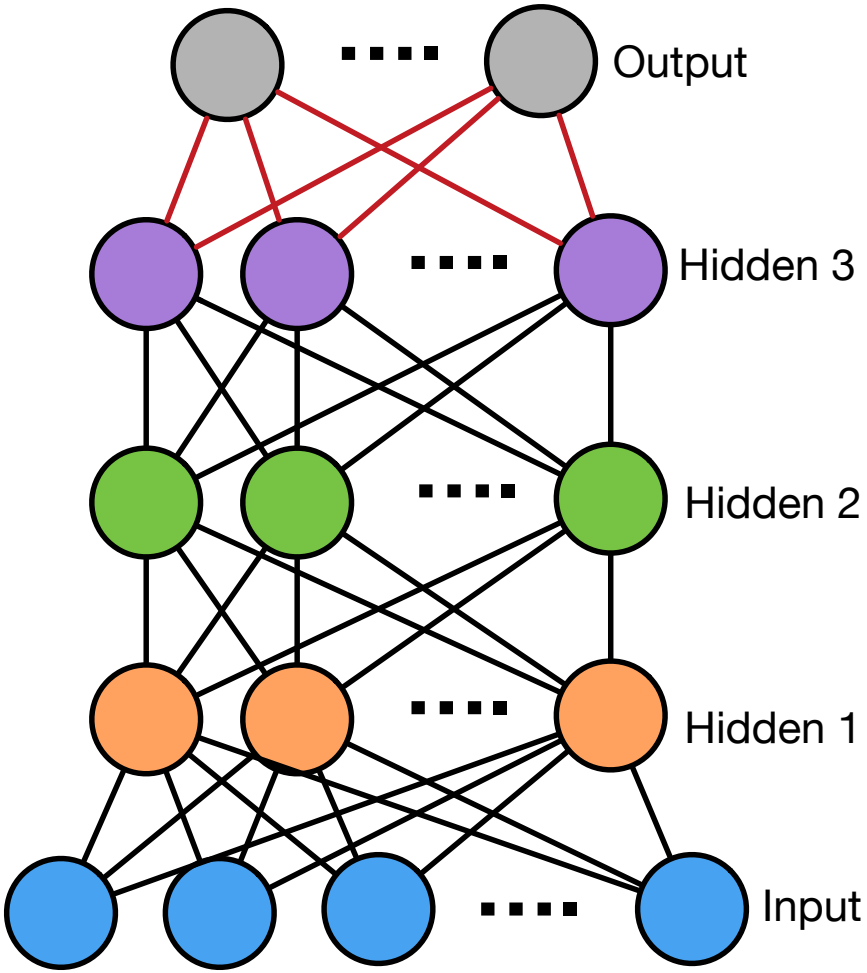


Pretraining using Stacked autoencoder



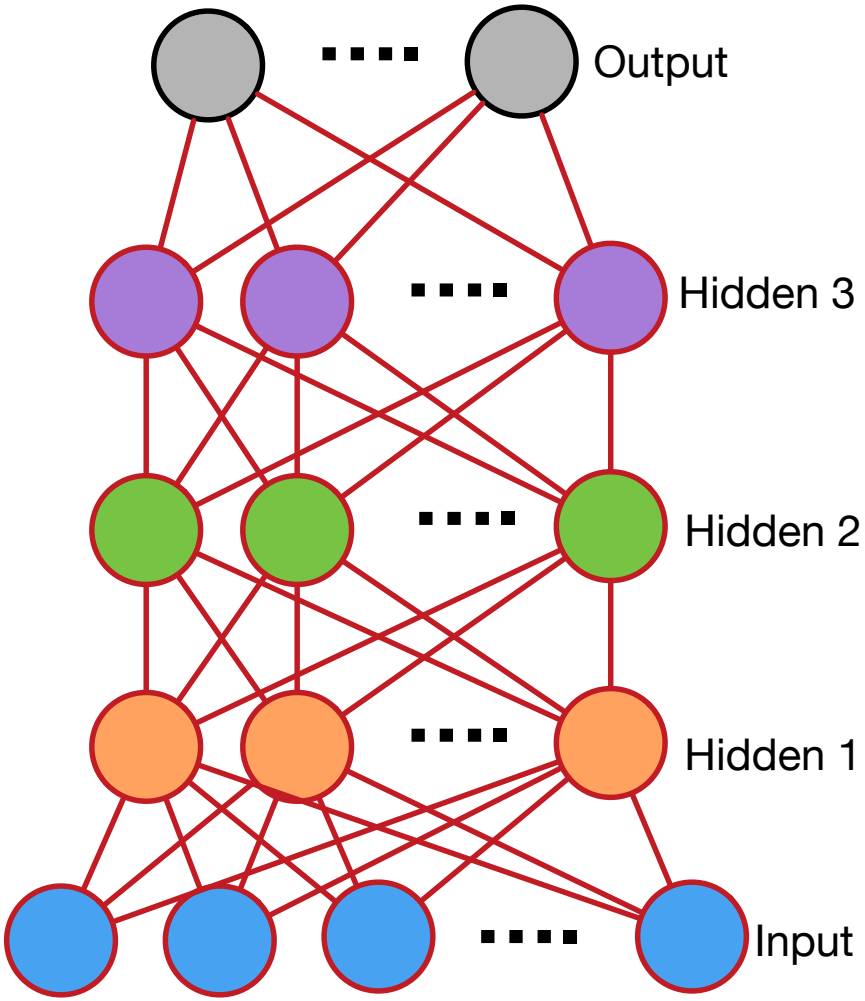
Initialise hidden layers

Pretraining using Stacked autoencoder



Train output layer

Pretraining using Stacked autoencoder

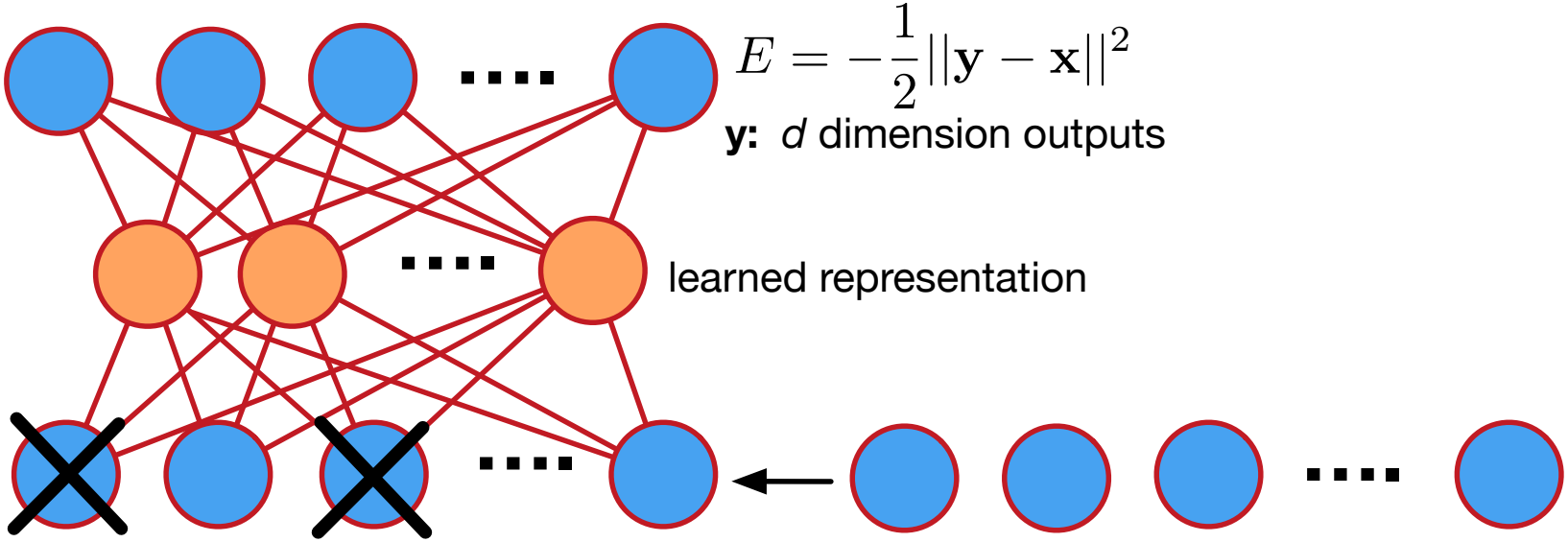


Fine tune whole network

Denoising Autoencoders

- Basic idea: Map from a corrupted version of the input to a clean version (at the output)
- Forces the learned representation to be stable and robust to noise and variations in the input
- To perform the denoising task well requires a representation which models the important structure in the input
- The aim is to learn a representation that is robust to noise, not to perform the denoising mapping as well as possible
- Noise in the input:
  - Random **Gaussian** noise added to each input vector
  - **Masking** – randomly setting some components of the input vector to 0
  - **"Salt & Pepper"** – randomly setting some components of the input vector to 0 and others to 1
- Stacked denoising autoencoders – noise is only applied to the input vectors, not to the learned representations

Denoising Autoencoder



Summary

- Feature normalisation
- Random parameter initialisation
- Batch normalisation
- Layer-by-layer Pretraining and Autoencoders
  - For many tasks (e.g. MNIST) pre-training seems to be necessary / useful for training deep networks
  - For some tasks with very large sets of training data (e.g. speech recognition) pre-training may not be necessary
  - (Can also pre-train using stacked restricted Boltzmann machines)
- Reading: Michael Nielsen, chapter 5 of *Neural Networks and Deep Learning*

  http://neuralnetworksanddeeplearning.com/chap5.html

  Pascal Vincent et al, "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion", JMLR, 11:3371–3408, 2010.

  http://www.jmlr.org/papers/volume11/vincent10a/