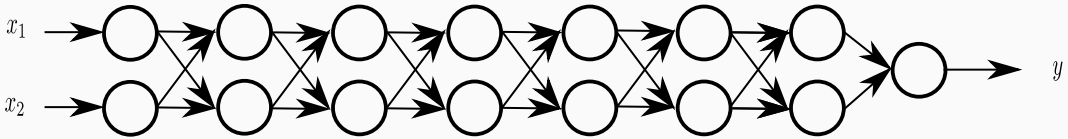
Initialisation, Normalisation, Dropout

Hakan Bilen

Machine Learning Practical — MLP Lecture 6

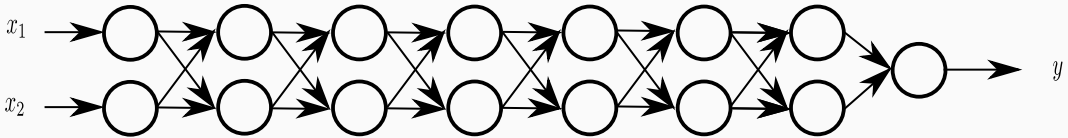
22 October 2019

Recap: Vanishing/exploding gradients



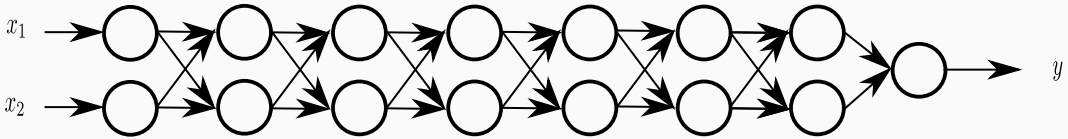
- $z^{(1)} = W^{(1)}x, \quad h^{(1)} = f(z^{(1)})$ and $y = h^{(L)}$
- Assuming f is identity mapping, $y = W^{(L)}W^{(L-1)} \dots W^{(2)}W^{(1)}x$

Recap: Vanishing/exploding gradients



- $z^{(1)} = W^{(1)}x, \quad h^{(1)} = f(z^{(1)})$ and $y = h^{(L)}$
- Assuming f is identity mapping, $y = W^{(L)}W^{(L-1)} \dots W^{(2)}W^{(1)}x$
- $W^{(l)} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \rightarrow \quad y = W^{(L)} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}^{L-1} x$ (Exploding gradients)

Recap: Vanishing/exploding gradients



- $z^{(1)} = W^{(1)}x, \quad h^{(1)} = f(z^{(1)})$ and $y = h^{(L)}$
- Assuming f is identity mapping, $y = W^{(L)}W^{(L-1)} \dots W^{(2)}W^{(1)}x$
- $W^{(l)} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \quad \rightarrow \quad y = W^{(L)} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}^{L-1} x$ (Exploding gradients)
- $W^{(l)} = \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix} \quad \rightarrow \quad y = W^{(L)} \begin{bmatrix} .5 & 0 \\ 0 & .5 \end{bmatrix}^{L-1} x$ (Vanishing gradients)

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

Setting $\mathrm{Var}(w)$

Consider a linear unit:

$$y = \sum_{i=1}^{n_{in}} w_i x_i$$

if w and x are zero-mean and iid (independent and identically distributed), then

$$\mathrm{Var}(y) = \mathrm{Var}(\sum_{i=1}^{n_{in}} w_i x_i) = n_{in}\,\mathrm{Var}(x)\,\mathrm{Var}(w)$$

So, if we want variance of inputs x and outputs y to be the same, set

$$\mathrm{Var}(w_i) = \frac{1}{n_{in}}$$

Nicely explained at http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization

"GlorotInit" ("Xavier initialisation")

We would like to set the variance of each layer to be $1/n_{in}$, thus

- Uniform distribution: $w_i \sim U(-\sqrt{3/n_{in}}, \sqrt{3/n_{in}})$. Hint $x \sim U(a, b)$, then $\text{Var}(x) = (b-a)^2/12$.
- However we need to take the backprop into account, hence we would also like $\text{Var}(w_i) = 1/n_{out}$
- As a compromise set the variance to be $\text{Var}(w_i) = 2/(n_{in} + n_{out})$
- This corresponds to Glorot and Bengio's normalised initialisation

$$w_i \sim U\left(-\sqrt{6/(n_{in} + n_{out})}, \sqrt{6/(n_{in} + n_{out})}\right)$$

"GlorotInit" ("Xavier initialisation")

We would like to set the variance of each layer to be $1/n_{in}$, thus

- Uniform distribution: $w_i \sim U(-\sqrt{3/n_{in}}, \sqrt{3/n_{in}})$. Hint $x \sim U(a, b)$, then $\text{Var}(x) = (b-a)^2/12$.
- However we need to take the backprop into account, hence we would also like $\text{Var}(w_i) = 1/n_{out}$
- As a compromise set the variance to be $\text{Var}(w_i) = 2/(n_{in} + n_{out})$
- This corresponds to Glorot and Bengio's normalised initialisation

$$w_i \sim U\left(-\sqrt{6/(n_{in}+n_{out})}, \sqrt{6/(n_{in}+n_{out})}\right)$$

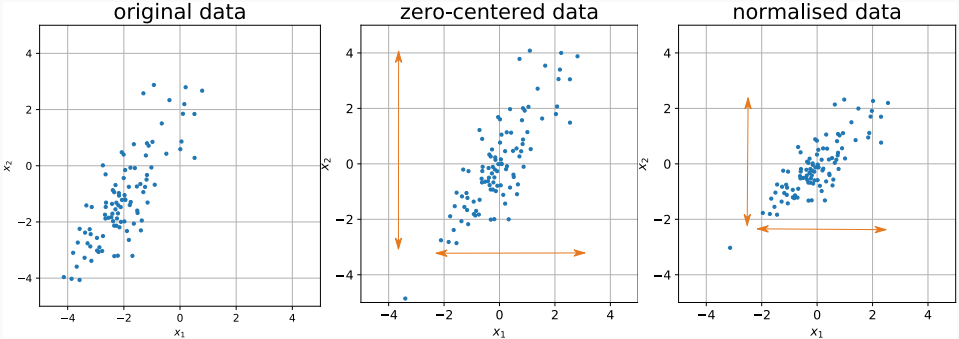
- For Normal distribution: $w_i \sim \mathcal{N}(0, \sqrt{1/n_{in}})$ and $w_i \sim \mathcal{N}\left(0, \sqrt{2/(n_{in}+n_{out})}\right)$.

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

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

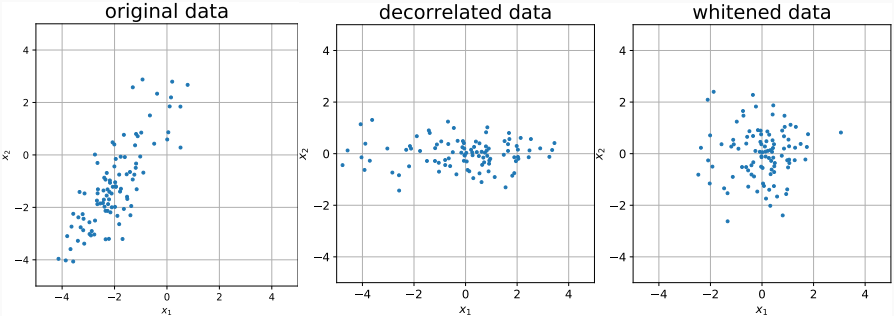
Feature Normalisation

Data Preprocessing – Normalization



- **Mean substraction**: $\hat{x}_i = x_i - \text{mean}(X^{train})$
- **Normalisation**: $\tilde{x}_i = \hat{x}_i / \text{std}(X^{train})$
- We also need to normalise test set with the train set mean and std

Data Preprocessing – PCA and Whitening

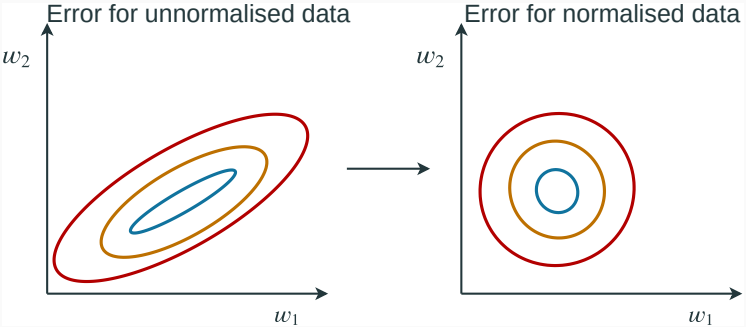


PCA (Principal Components Analysis)

- Decorrelate the data by projecting onto the principal components.
- Also possible to reduce dimensionality by only projecting onto the top P principal components.

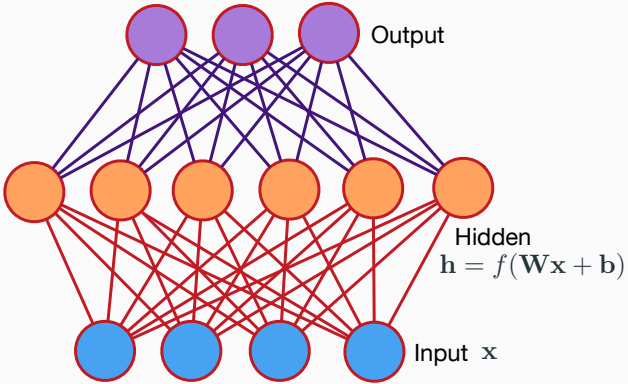
Whitening: PCA + Scale each dimension

Why is whitening useful?



Normalising input can result in faster training by enabling training with higher learning rates.

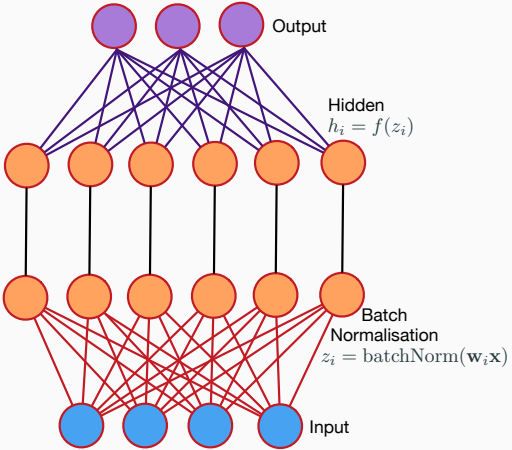
Batch Normalisation



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

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

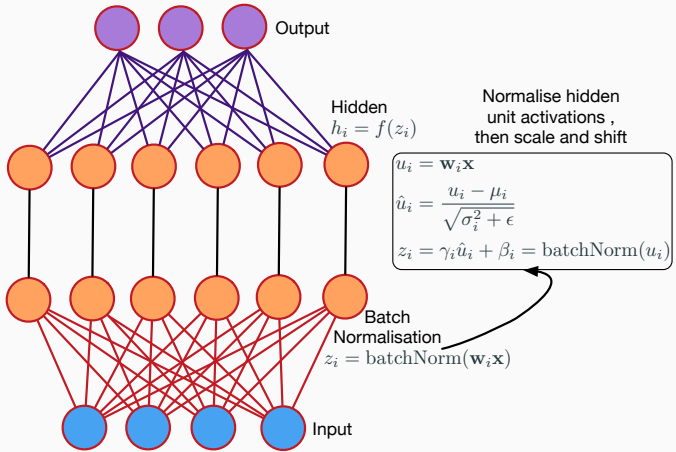
Batch Normalisation



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

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

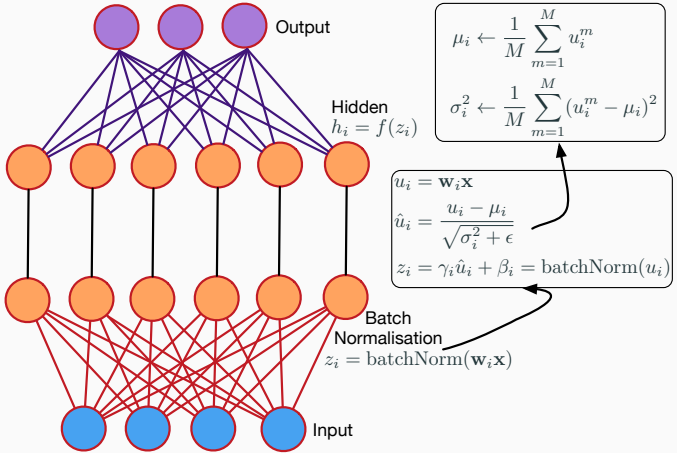
Batch Normalisation



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

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

Batch Normalisation

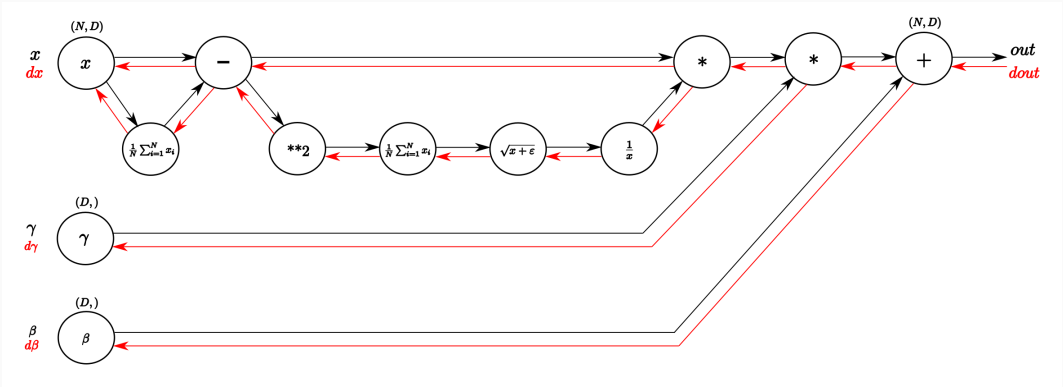


Compute mean and variance of each hidden unit activation across the minibatch (size M)

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

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

Computational graph for batch normalisation



https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.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

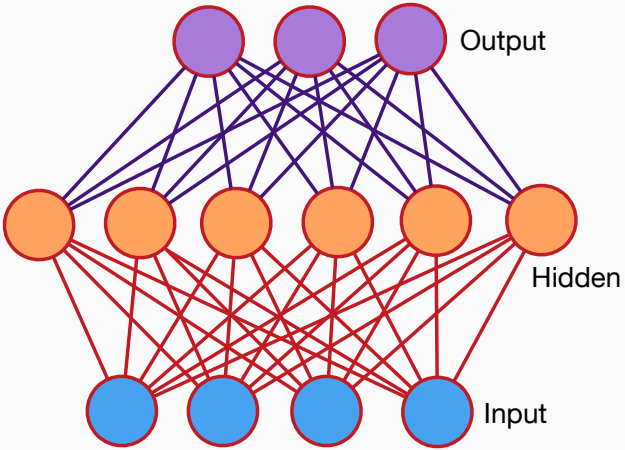
- Makes training many-layered networks easier
  - Allows higher learning rates
  - Weight initialisation less crucial
- Can act like a regulariser – maybe reduces need for techniques like dropout
- Can be applied to convolutional networks
- In practice (image processing) – achieves similar accuracy with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

Dropout

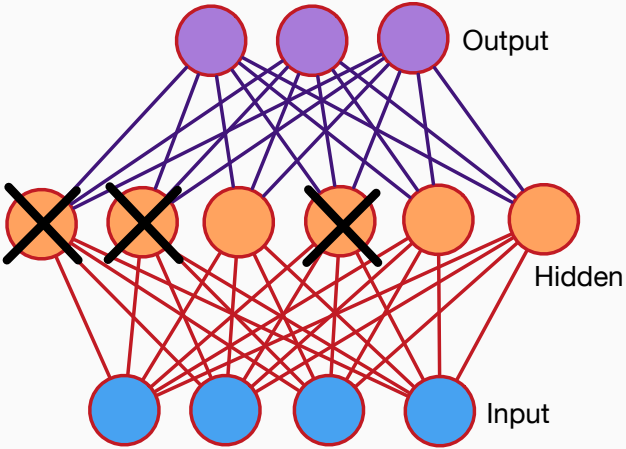
Dropout

- **Dropout** is a way of training networks to behave so that they have the behaviour of an average of multiple networks
- Dropout training:
  - Each mini-batch randomly delete a fraction of the hidden units (inclusion probability $p \sim 0.5$) and the input units ($p \sim 0.8$) – and their related weights and biases
  - Then process the mini-batch (forward and backward) using this modified network, and update the weights
  - Restore the deleted units/weights, choose a new random subset of hidden units to delete and repeat the process

Dropout Training - Complete Network

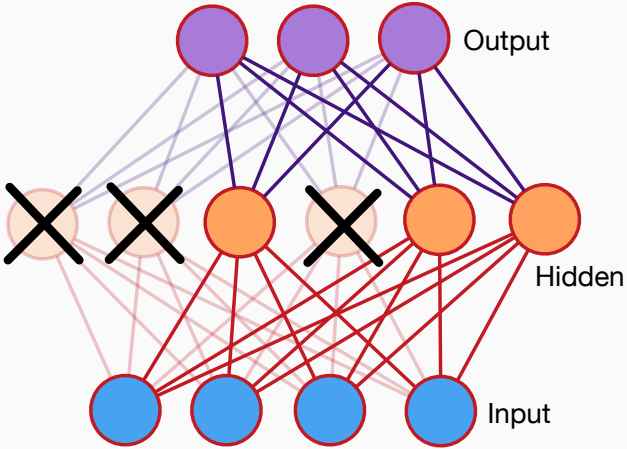


Dropout Training - First Minibatch



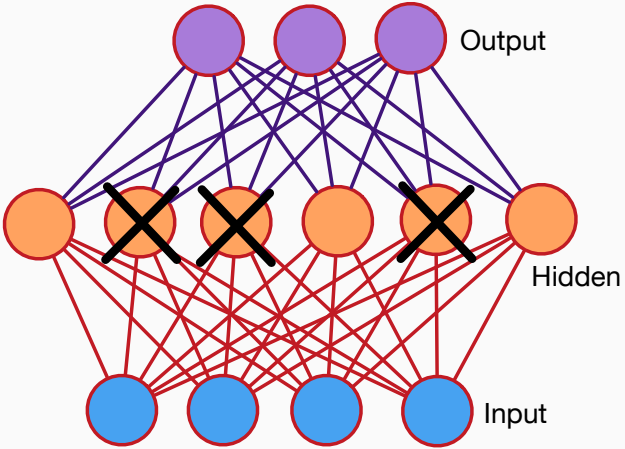
$p = 0.5$

Dropout Training - First Minibatch



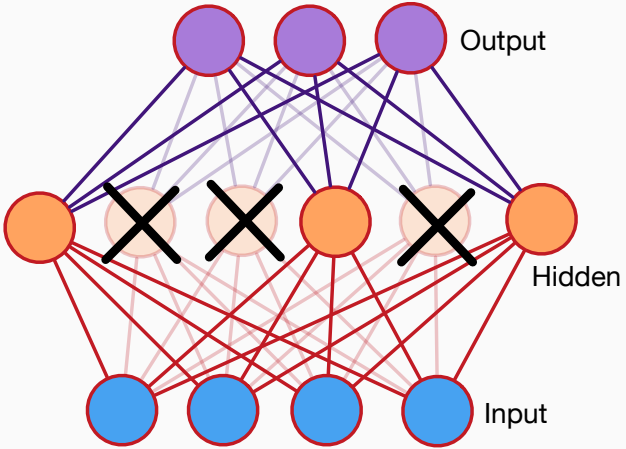
$p = 0.5$

Dropout Training - Second Minibatch



$p = 0.5$

Dropout Training - Second Minibatch



$p = 0.5$

Dropout

- **Dropout** is a way of training networks to behave so that they have the behaviour of an average of multiple networks
- Dropout training:
  - Each mini-batch randomly delete a fraction of the hidden units (inclusion probability $p \sim 0.5$) and the input units ($p \sim 0.8$) – and their related weights and biases
  - Then process the mini-batch (forward and backward) using this modified network, and update the weights
  - Restore the deleted units/weights, choose a new random subset of hidden units to delete and repeat the process

Dropout

- **Dropout** is a way of training networks to behave so that they have the behaviour of an average of multiple networks
- Dropout training:
  - Each mini-batch randomly delete a fraction of the hidden units (inclusion probability $p \sim 0.5$) and the input units ($p \sim 0.8$) – and their related weights and biases
  - Then process the mini-batch (forward and backward) using this modified network, and update the weights
  - Restore the deleted units/weights, choose a new random subset of hidden units to delete and repeat the process
- To compensate for missing units, use "inverted dropout": scale by $1/p$ when training, no scaling in final network.

Why does Dropout work?

- Each mini-batch is like training a different network, since we randomly select to dropout (remove) a fraction of the units
- So we can imagine dropout as combining an exponential number of networks
- Since the component networks will be complementary and overfit in different ways, dropout is implicit model combination
- Also interpret dropout as training more robust hidden unit features – each hidden unit cannot rely on all other hidden unit features being present, must be robust to missing features
- Dropout has been useful in improving the generalisation of large-scale deep networks
- **Annealed Dropout:** Dropout rate schedule starting with a fraction p units dropped, decreasing at a constant rate to 0
  - Initially training with dropout
  - Eventually fine-tune all weights together

Lab 6: 06_Dropout_and_Maxout

Lab 6 explores dropout:

- Implementing a Dropout Layer
- Training models with dropout layers to classify MNIST digits

The lab also explores another non-linear transformation, Maxout, which can be thought of as a generalisation of ReLU

- Implementing Maxout using a Max Pooling Layer
- Training models with maxout layers to classify MNIST digits

Summary

- Initialisation – how to intitialise the weights, independent of network size
- Batch normalisation – normalise activations of each layer
- Dropout – train networks so they behave as an average of multiple networks
- Additional material: 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)

Mid-semester survey

Please take 5 minutes to complete the mid-semester survey

https://edinburgh.onlinesurveys.ac.uk/machine-learning-practical-201920-semester-1-mid-semes

Link is also at top of the course homepage

Reading

- Michael Nielsen, chapter 5 of *Neural Networks and Deep Learning*
  http://neuralnetworksanddeeplearning.com/chap5.html
- Goodfellow et al, sections 7.12, 8.4, 8.7.1, chapter 14
- Additional reading:
  - Srivastava et al, "Dropout: a simple way to prevent neural networks from overfitting", JMLR, 15(1), 1929-1958, 2014.
    http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf
  - Glorot and Bengio, "Understanding the difficulty of training deep feedforward networks", AISTATS-2010.
    http://www.jmlr.org/proceedings/papers/v9/glorot10a.html
  - Ioffe and Szegedy, "Batch normalization", ICML-2015.
    http://www.jmlr.org/proceedings/papers/v37/ioffe15.html
  - Kratzert, "Understanding the backward pass through Batch Normalization Layer".
    https://kratzert.github.io/2016/02/12/

Additional Material
Pretraining and Autoencoders

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

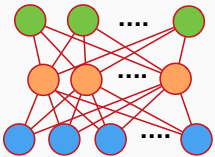
- Supervised: Layer-by-layer cross-entropy training
- Unsupervised: Autoencoders
- Unsupervised: Restricted Boltzmann machines (not covered in this course)

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

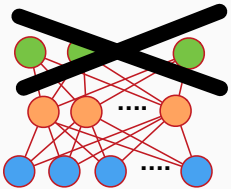
Greedy Layer-by-layer cross-entropy training

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



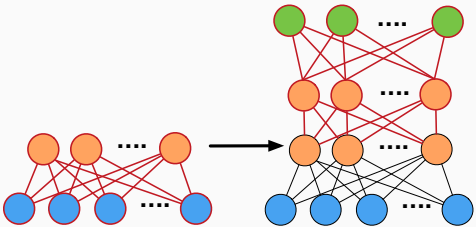
Greedy Layer-by-layer cross-entropy training

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



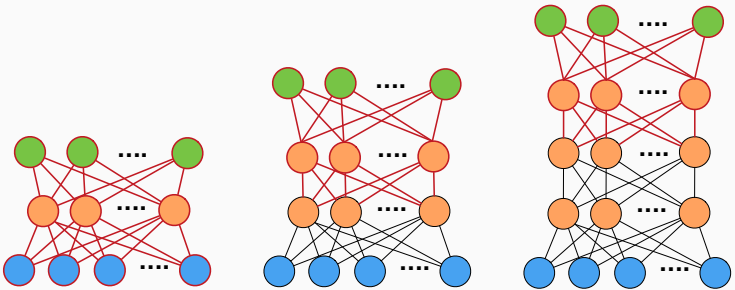
Greedy Layer-by-layer cross-entropy training

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



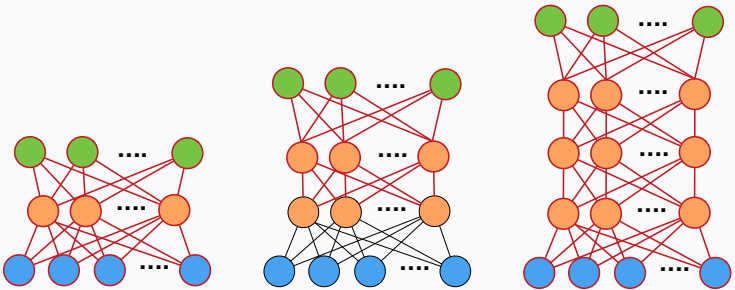
Greedy Layer-by-layer cross-entropy training

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



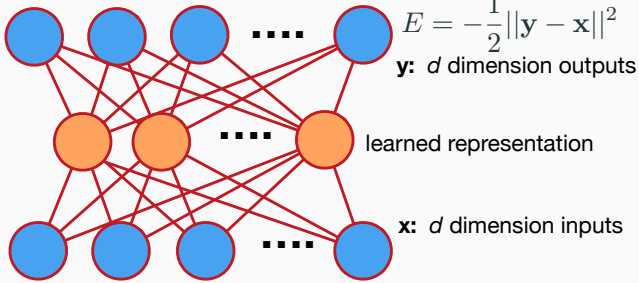
Greedy Layer-by-layer cross-entropy training

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



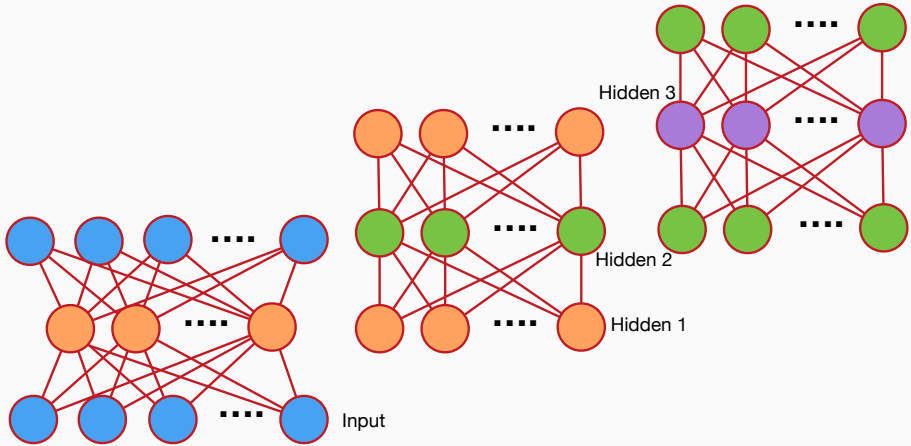
Stacked autoencoders

- 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

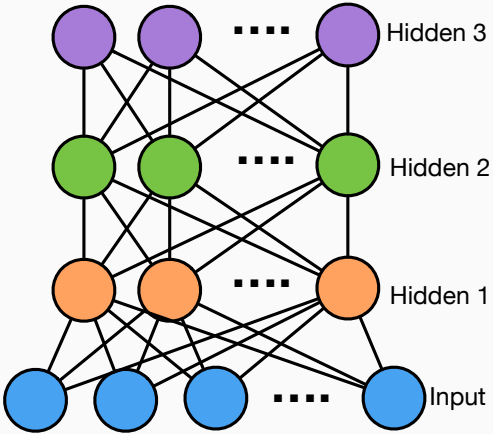
Stacked autoencoders

- 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

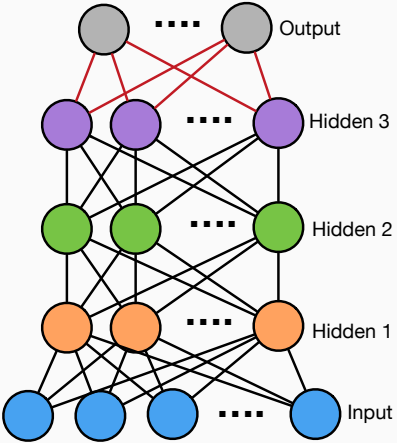


Pretraining using Stacked autoencoder



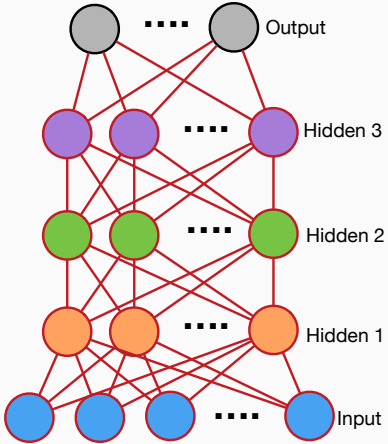
Initialise hidden layers

Pretraining using Stacked autoencoder



Train output layer

Pretraining using Stacked autoencoder

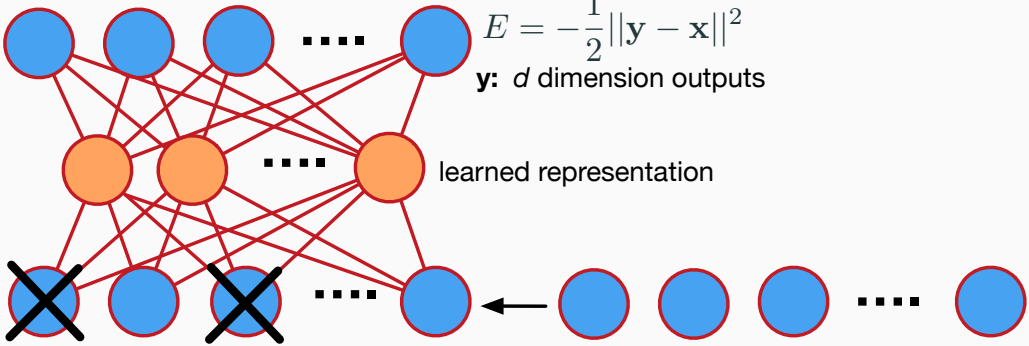


Fine tune whole network

Denoising Autoencoders

- Basic idea: Map a corrupted input to a clean 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



Lab 7: 07_Autoencoders

Lab 7 explores autoencoders and pretraining:

- Implementing a linear autoencoder
- Implementing a non-linear autoencoder
- Denoising autoencoders
- Using an autoencoder as an initialisation for supervised training