Artificial Neural Networks & Deep Learning: Report

September 17, 2025

Contents

1 Supervised Learning and Generalisation **1**
- 1.1 The Perceptron . . . 1
- 1.2 Backprop . . . 1
- 1.3 Training Algorithms . . . 2
- 1.4 Regression . . . 4

2 Recurrent Neural Networks (RNNs) **5**
- 2.1 Hopfield Networks . . . 5
- 2.2 Time Series Prediction . . . 7

3 Deep Feature Learning **9**
- 3.1 Image Reconstruction with Autoencoders . . . 9
- 3.2 Image Classification with Stacked Autoencoders . . . 9
- 3.3 Convolutional Neural Networks (CNNs) . . . 10
- 3.4 Attention and Transformers . . . 11

4 Generative Models **13**
- 4.1 Energy-Based Models . . . 13
- 4.2 Generative Adversarial Networks (GANs) . . . 15
- 4.3 Variational Autoencoders (VAEs) . . . 16

A Delta Rule **17**

B Backprop Example **19**

C Google Research Tuning Playbook **22**

D Review Questions **23**

1 Supervised Learning and Generalisation

1.1 The Perceptron

1.1.1 Linear Regression with a Perceptron

A perceptron with linear activation (1) and MSE is equivalent to linear regression (2):

$$y \approx f(x; w_1, \dots, w_N, b) = \sigma\big(\sum_{i=1}^{N} w_i x_i + b\big),\ \sigma \in \{\mathit{linear}\} \notin \{\tanh, \mathit{sigmoid}, \mathit{sign}, \dots\} \quad (1)$$

$$y \approx f(x; w_1, \dots, w_N, b) = \sum_{i=1}^{N} w_i x_i + b, \quad \arg\min_{w_1,\dots,w_N,b} \big(y - f(x; w_1, \dots, w_N, b)\big)^2 \quad (2)$$

1.1.2 Linear Models

A linear model $\sum_{i=1}^{N} w_i x_i + b$ is *linear* in the parameters [2] (and bias) $w_1, \dots, w_N, b$, but may be nonlinear in the features, e.g. polynomial regression with features $x, x^2, \dots,$ or some basis functions - which would be able to capture sinusoidal data (perfectly).

A perceptron only captures linear relationships; and underfits sinusoidal data. A MLP with nonlinear activations and multiple layers of (a sufficient number of) hidden neurons, however, e.g. $f = W\sigma\big(H_2\sigma(H_1 x + b_1) + b_2\big)$, is a universal function approximator. [11]

Moreover, the approximation error for an MLP, $\mathcal{O}(\frac{1}{n_h})$, depends on the no. of hidden units but *not* on the input space dimension n - unlike for polynomial expansions, $\mathcal{O}(\frac{1}{n_p^{\frac{2}{n}}})$.

1.1.3 The Effect of the Learning Rate

A small learning rate η causes gradient descent to take smaller steps, while a large η causes it to take bigger steps. Consequently, with a smaller learning rate, training loss decreases more slowly and gradient descent converges more slowly; if η is too large it may not converge. The experiments validate this empirically, e.g. with $\eta = 0.002, 0.2, 2.0$.

Learning rate decay[1] [29] and adaptive optimisers like Adam [13] (larger updates for parameters with smaller gradients, smaller updates for parameters with larger gradients) adapt the learning rate during training - with the aim of speeding up convergence.

1.1.4 The Loss Curve

With a larger learning rate, mini-batches give a noisy estimate[2] of the gradient for SGD. Adam's adaptive learning rates result in some larger or smaller steps.

1.2 Backprop

Backpropagation recursively propagates the error from the output back to the input.

[1] An alternative to learning rate decay / scheduling is to gradually increase the batch size. [23] Increasing the batch size reduces gradient noise, which similarly facilitates optimisation. Although in general, the batch size cannot be changed blindly as it alters the training dynamics, and requires tuning hyperparameters like effective learning rate and regularisation accordingly. [8]

[2] SGD updates are noisy but unbiased; i.e. convergence in expectation - with a suitable learning rate.

1.3 Training Algorithms

1.3.1 The Effect of Noise on the Optimisation Process

"Injecting noise . . . [is] a form of data augmentation" [22, 9] although "neural networks prove not to be very robust to noise". [24, 9] The model still achieves good training and validation loss values after 15 to 25 epochs. When trained on pure noise, loss remains high (i.e. not learning much signal); from 20 epochs it starts overfitting the noise somewhat.

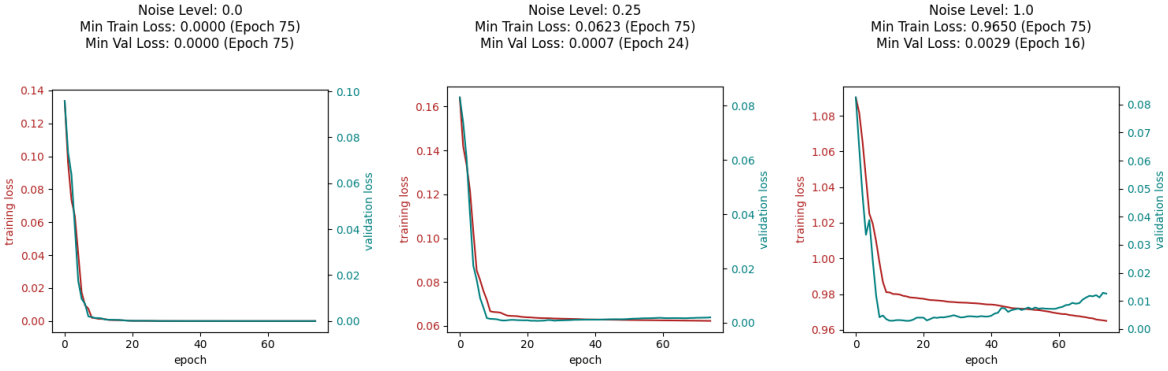

Figure 1: L-BFGS training and validation loss curves with 0 vs. 25 vs. 100% noise

Noise, e.g. $\epsilon \sim \mathcal{N}(0, \sigma^2), \epsilon \perp g$, affects the gradients, $\tilde{g} = g + \epsilon$: unbiased, $\mathbb{E}[\tilde{g}] = g$, but higher *gradient* variance, $Var(\tilde{g}) = Var(g) + Var(\epsilon)$. Some noise may act as regularisation *and* improve optimisation [15] with gradient variance often yielding better generalisation, avoiding local minima; too much noise makes training unstable (Figure 1.3.2, cf. SGD[3]).

1.3.2 GD vs. SGD vs. AGD vs. Adam vs. L-BFGS

Full-batch gradient descent is guaranteed to decrease the loss on every iteration with a sufficiently small learning rate, i.e. $\eta < 2/L$, for a Lipschitz-smooth convex function. Stochastic mini-batch gradient descent exhibits stochastic gradient updates based on a mini-batch of the full data and oscillates (more, depending on the batch size). Adaptive methods and second-order approximations are designed to accelerate convergence: Adam [13], RMSProp [10] and Adadelta [30] do so by adapting the learning rate using gradient moments, or running averages of past gradients, respectively. NAG [16] is momentum-based with a look-ahead strategy.[4] Limited-memory BFGS [1] is quasi-Newton: it approximates the Hessian & uses line search to determine optimal step size.

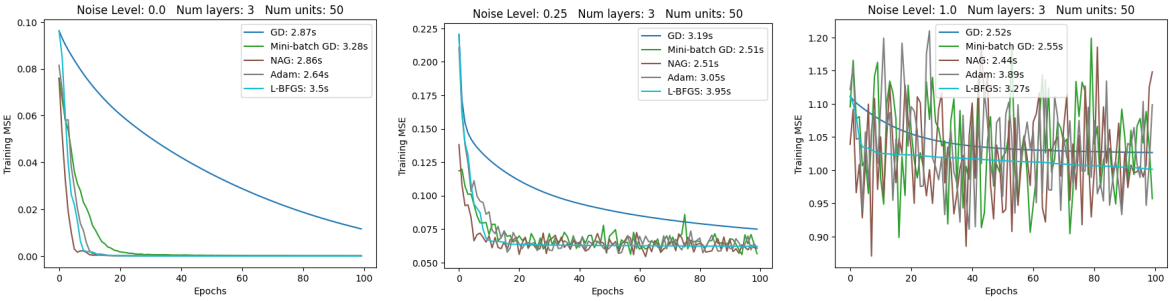

Figure 2: GD, mini-batch GD, NAG, Adam, L-BFGS loss with 0 vs. 25 vs. 100% noise

[3]Small batches, esp. SGD, often act as regularisation at the cost of stability, runtime (smaller η). [9]
[4]Loss spikes in the first epochs may require learning rate warm-up, or reducing batch size.

1.3.3 Impact of Network Size on the Choice of Optimizer

The larger the size of the network, the better Adam and mini-batch GD perform compared with NAG and L-BFGS. Also full-batch GD performs better for larger networks.

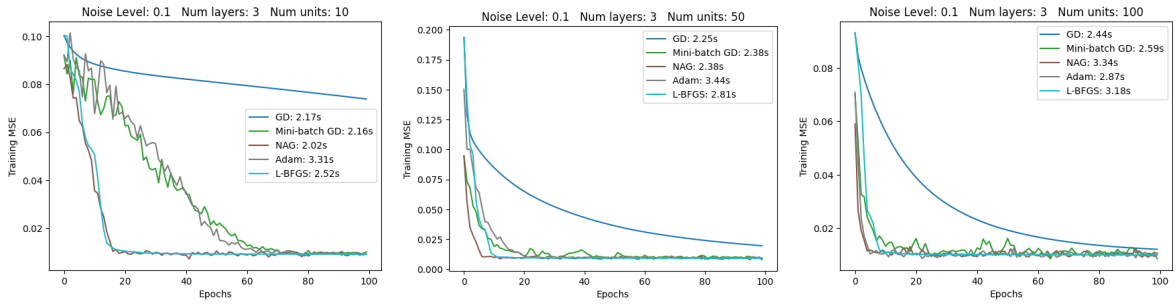

Figure 3: GD, mini-batch GD, NAG, Adam, L-BFGS loss with 10 vs. 50 vs. 100 neurons

1.3.4 Epochs, Time and Convergence

An epoch is one pass through the training data, with $1 \leq |\text{batch}| \leq |\text{training data}|$. Time is the actual time taken, depending on the optimiser, hardware, etc., and on the batch size. Convergence occurs at the point where the objective stabilises and/or does not improve further. Speed of convergence may thus be in terms of no. of epochs or time. In practice it is training *time* that matters (and computation, cf. e.g. Newton's method).

GD shows poor convergence speed in time and no. of epochs. The other algorithms converge well in time, no. of epochs, although mini-batch GD may require more epochs.

1.3.5 Parametrisation of a Convolutional Neural Network (CNN)

The CNN comprises two convolutional blocks (each with a conv and max pooling layer), followed by a flattening, dropout, and dense output layer. Each conv layer uses a kernel with specified height, width, input channels, and output filters, plus one bias per filter. Pooling layers downsample the feature maps. The CNN has 34,826 trainable[5] parameters.

The first convolution receives input images of shape $28 \times 28 \times 1$, uses a 3×3 kernel, and outputs 32 filters: $(3 \times 3 \times 1 \times 32) + 32 = 320$ parameters (288 weights, 32 biases). Both pooling layers are 2×2. After pooling, the second convolution operates on input of shape $13 \times 13 \times 32$, with the same kernel and 64 filters: $(3 \times 3 \times 32 \times 64) + 64 = 18{,}496$ parameters. The output is flattened, a dropout of 0.5 is applied, and a final dense softmax layer maps 1600 inputs to 10 outputs, totaling $(1600 \times 10) + 10 = 16{,}010$ params. [6, 31]

1.3.6 Training a CNN on MNIST with Adam vs. SGD vs. Adadelta

With early stopping [6] and restoring best weights, Adam achieves ≈ 0.9929 test accuracy (and ≈ 0.0217 test loss) in 30 epochs, SGD (with learning rate scheduling) ≈ 0.9909 (and ≈ 0.0263) in 43 epochs, and Adadelta (with tuned learning rate) ≈ 0.9888 (and ≈ 0.0345) in 50 epochs. Further hyperparameter tuning could further improve performance.

Adadelta [30] (like RMSProp [10]) is a variant of Adagrad [5], computing per-parameter learning rates, but adapting them less aggressively with "squared *rescaled* gradients". [31]

[5]Non-trainable parameters appear in layers like frozen layers (e.g., when reusing pre-trained embeddings) and batch normalisation. A batch normalisation layer — insertable between conv and pooling — would add trainable scale and shift parameters, and non-trainable running mean and variance.

[6]Early stopping is performed on validation (categorical cross-entropy) loss, with continuous gradient, decreases more smoothly and gives a more stable signal of generalisation than discrete accuracy.

1.4 Regression

Training and Test Split The dataset is shown in Figure 4. It is split into a training (second from left) and a test (first from right) set by randomly sampling without replacement s.t. no point is in both. Such an independent, identically distributed test set (i.i.d., i.e. sampled from the same distribution but may not overlap with the training data) is needed to evaluate a model's capabilities of generalisation to unseen, out-of-sample data.

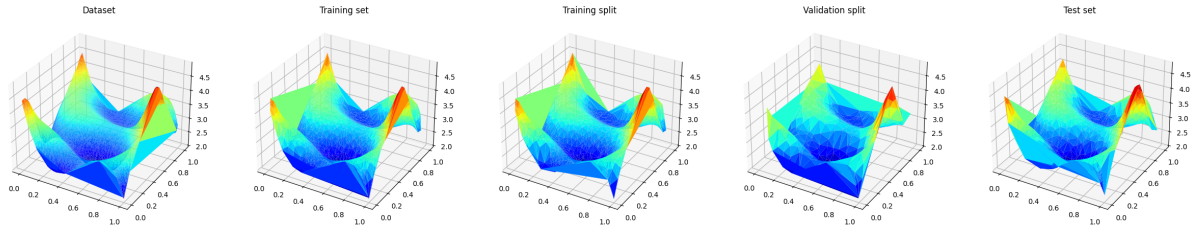

Figure 4: Dataset, training set (with training and validation splits), test set

Model Selection For model selection, a grid search is used, as the space is not too large: Adam with a fixed learning rate is used as a default to run a grid search on the number of layers and neurons and activation function; finally learning rate, optimiser, etc. are tuned. To validate the model, the training set from the initial split (second from left, in Figure 4) is split further, into a training set (third) and a validation set (fourth).

A shallow network (i.e. with one single hidden layer) and unbounded activations in theory is sufficient as a universal approximator for a continuous nonlinear function [20]. However, a deep network is more efficient (i.e. less complexity) at representing a function, e.g. by utilising hierarchical features [25] at the expense of potentially more time and difficulties (e.g. vanishing gradients) to train. Two layers with 20 neurons work well here.

Model Evaluation No further training is possible in validation-based model selection because, by definition, the model is selected based on the validation set and only evaluated on the test set. This prevents data leakage and enables evaluation on unseen data.

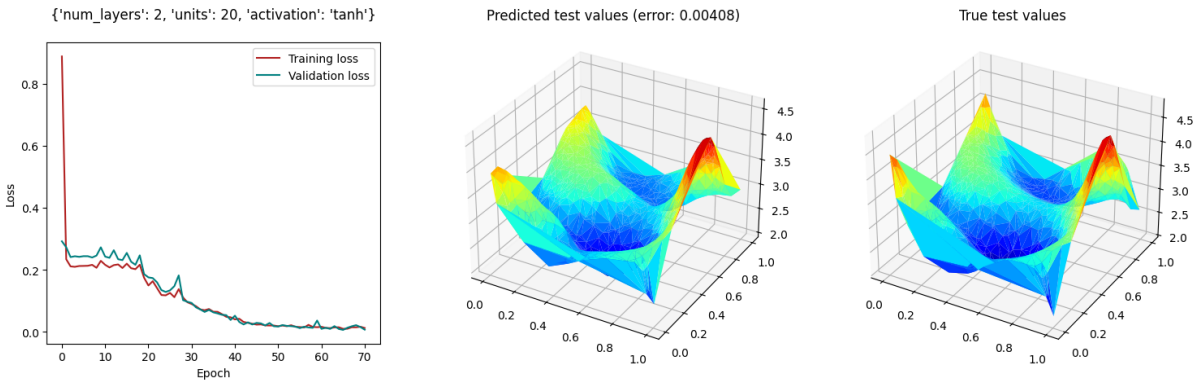

Figure 5: Training and validation loss curves, and prediction on test set (MSE: 0.004)

Overfitting and Regularisation Regularisation reduces a model's flexibility by penalising large or complex parameters, and thereby lowering its effective number of parameters or degrees of freedom. Here, early stopping is used to prevent overfitting - as evidenced on the test set (Figure 5). L1 / L2 and dropout are other common methods.

2 Recurrent Neural Networks (RNNs)

2.1 Hopfield Networks

2.1.1 A Two-Dimensional Network

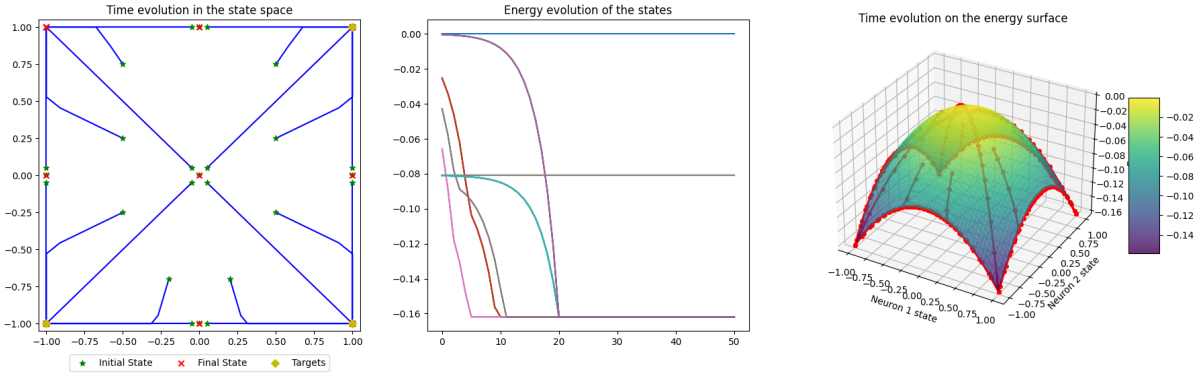

Figure 6: Evolution of 2D Hopfield Network over 50 iterations

Spurious Attractors The set of targets S used to create the network are all attractors: `[[-1. -1.] [ 1. -1.] [ 1.  1.]]`. In addition, there are spurious attractors due to symmetric minima (`[[-1. 1.]]`, Equations 4, 5), maxima (`[[ 0. 0.]]`) or saddle points (`[[1. 0.], [ 0. 1.], ...]`) on the energy surface. That is, symmetry gives rise to attractors also in points symmetric to or equidistant from the set of targets S.

$$W = \sum_{\boldsymbol{s} \in S} \boldsymbol{s}\boldsymbol{s}^T = \begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}^T + \begin{bmatrix}-1\\-1\end{bmatrix}\begin{bmatrix}-1\\-1\end{bmatrix}^T + \begin{bmatrix}1\\-1\end{bmatrix}\begin{bmatrix}1\\-1\end{bmatrix}^T = \begin{bmatrix}3 & 1\\1 & 3\end{bmatrix}, \quad \min_{\boldsymbol{s}} -\frac{1}{2}\boldsymbol{s}^T W \boldsymbol{s} \tag{3}$$

$$\boldsymbol{x}^{(n+1)} = \alpha(W\boldsymbol{x}^{(n)}), \qquad \alpha\big(W \cdot (-\boldsymbol{x}^{(n)})\big) = \alpha(-W\boldsymbol{x}^{(n)}) = -\alpha(W\boldsymbol{x}^{(n)}) = -\boldsymbol{x}^{(n+1)} \tag{4}$$

$$\alpha\left(W\alpha\left(W\begin{bmatrix}0.5\\-0.75\end{bmatrix}\right)\right) = \begin{bmatrix}1\\-1\end{bmatrix} \in S, \qquad \alpha\left(W\alpha\left(W\begin{bmatrix}-0.5\\0.75\end{bmatrix}\right)\right) = \begin{bmatrix}-1\\1\end{bmatrix} \notin S \tag{5}$$

Convergence In this network, it may take up to ≈ 20 iterations to reach an attractor.

Stability An unstable attractor is one such that if a pattern is initialised in an unstable attractor, it remains there. But just a small perturbation of the initial state would cause it to converge to a stable attractor instead.

Minima minimise energy, i.e. are stable (incl. spurious attractors arising due to symmetry). Maxima or saddle points do not minimise energy, i.e. are unstable.

Design To accurately store patterns or memories, a good associative memory network should *ideally* have no spurious, nor unstable, attractors.

2.1.2 A Three-Dimensional Network

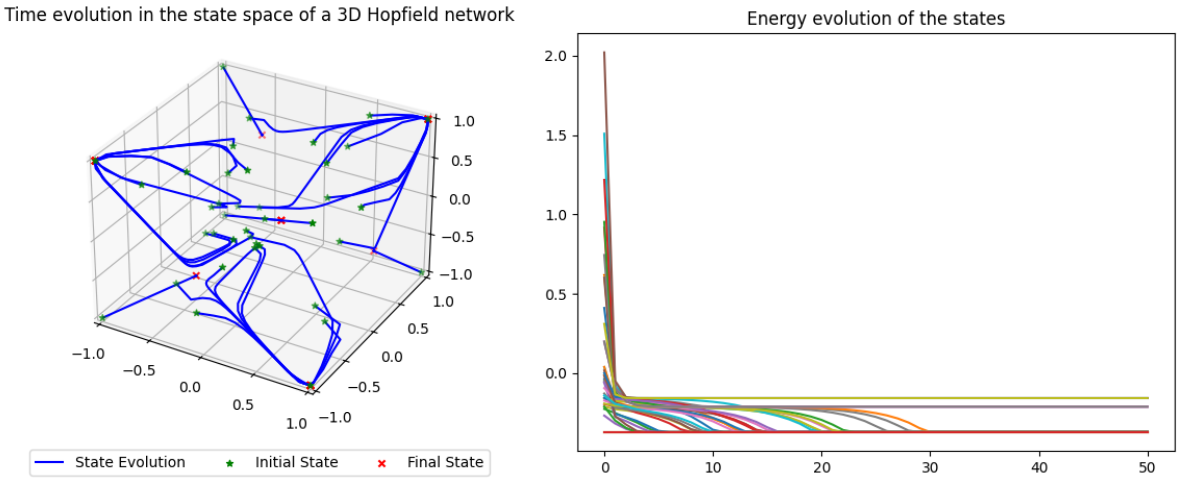

Figure 7: Evolution of 3D Hopfield Network over 50 iterations

Spurious Attractors The set of targets S used to create the network are all attractors: `[[-1. -1.  1.] [ 1. -1. -1.] [ 1.  1.  1.]]`. In addition, there are spurious attractors. These are the midpoints on the edges between the targets: `[[-0.06 -1. -0.06] [-0.06 0.06 1.] [ 1. 0.06 -0.06]]` $\left(\left(\frac{-1+1}{2}, \frac{-1-1}{2}, \frac{1-1}{2}\right) = \left(0, -1, 0\right), \text{etc.}\right)$. And the circumcentre of the triangle formed by the targets: `[[ 0.367 -0.367  0.367]]`.

Convergence Like in two dimensions, in most cases it takes $<\approx 20$ iterations to reach an attractor, but may take up to 30 or 40 iterations in a three-dimensional network.

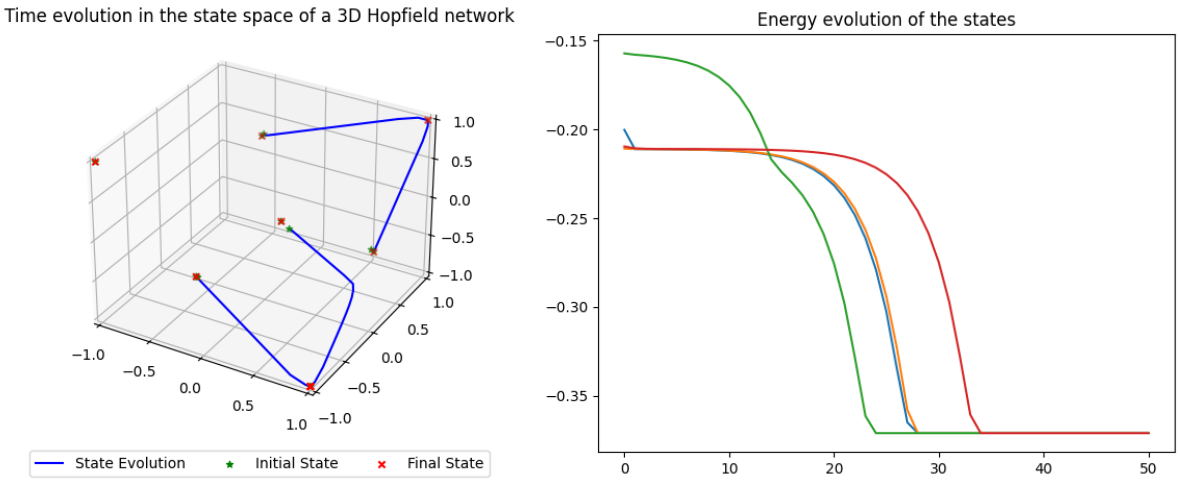

Figure 8: Evolution of 3D Hopfield Network over 50 iterations near spurious attractors

Stability Initial points near spurious attractors do not converge to the spurious attractors, i.e. spurious states may be unstable. An analytical solution could confirm this.

2.1.3 Digit Recognition with a Higher-Dimensional Network

A higher-dimensional Hopfield Network can be used to store handwritten digits. Retrieval from the network can be tested by having it reconstruct digits corrupted by noise.

Reconstruction Capacity The Hopfield Network performs very well - better than the untrained human eye - but is *not always able to reconstruct the noisy digits*. This is because the additive noise changes the image. That is, with noise the image's initial state will be farther away from the intended attractor. If there is too much noise it does not look like the original digit anymore and may even become more similar (closer) to a different digit (attractor). In such a case the network produces an incorrect reconstruction.

For noise levels of 0%, 10%, 20%, 30% and even 40% the network is always able to reconstruct all digits correctly in the experiments. From 50% noise the network begins to make mistakes. In particular, it makes mistakes between digits with similar patterns, like threes, fives and nines. It also mistakes an eight for a one, likely because the eight is very narrow in the dataset, which with noise could look like and converge to a one.

Number of Iterations *The more noise* there is the farther away from their intended attractors the noisy digits will be and *the more iterations it takes* for the network to (fully) reconstruct the given digits (i.e. converge to the attractors).

For noise levels of 0%, 10%, 20%, 30% the network is able to denoise all the correct digits after just one iteration so that the human eye can recognise them. For full reconstruction (convergence to the attractors) it takes up to $\approx 5, 10, 20$ iterations respectively in the experiments, however. With 40% noise after just one iteration it is not possible to decipher anything yet, although after 25 iterations the correct attractors are reached.

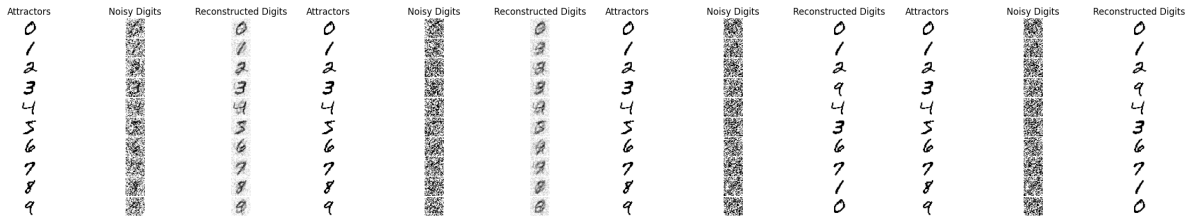

Figure 9: Digit reconstruction: (a) 20% noise/1 it., (b) 40%/1, (c) 40%/25, (d) 50%/50

2.2 Time Series Prediction

A dynamical system evolves over time, often following physical rules. The Santa Fe laser data, with 1000 training and 100 test steps, behaves nonlinearly due to e.g. delay, noise.

2.2.1 Multi-Layer Perceptron (MLP) with One Hidden Layer

Two hyperparameters are systematically tuned for a single-hidden-layer MLP: the number of input lags and the size of the hidden layer. For each setting, the model is trained using K-fold (time series) cross-validation[7] and its out-of-sample performance is recorded. The

[7]Cross-validation is a validation technique that makes better use of limited amounts of available training data, at the expense of being more computationally and time-intensive. If there are temporal (or spatial, etc.) correlations as in time series cross-validation, care has to be taken when splitting the data to respect chronological order and prevent data leakage.

training regime (optimiser, etc.) is fixed for a fair and simple comparison, exploring the influence of both input history and network capacity on time series prediction accuracy.

The MLP performs poorly with smaller lag values. Increasing the lag to 50 or 100 and the number of hidden units to 50 improves the performance and the model is able to better recognise the drop that occurs and after which the time series is reset.

Lag 100 & 50 hidden units give the best performance (MSE 2462.667) on the test set.

2.2.2 Long-Short-Term Memory (LSTM)

Recurrent neural networks (RNNs) are neural architectures with loops that allow information to persist across time steps. Long Short-Term Memory (LSTM) networks are a type of RNN designed to better learn *long-term* dependencies, which standard RNNs struggle with due to the vanishing gradient problem. LSTMs achieve this by maintaining a memory cell and using gating mechanisms to regulate how much - controlled by sigmoid activations $\in [0, 1]$ - at each time step information is input, forgotten, or output. Variants include LSTMs with peephole connections and Gated Recurrent Units (GRUs), the latter of which combines the input and forget gates into a single update gate. [18]

Increasing the lag for the LSTM also improves its performance, although the LSTM also performs well already with a smaller lag value. It is especially able to better recognise the unexpected drop / change in the series, also with a 10 times smaller lag. It is important to also tune the number of hidden units. 100 hidden units are sufficient to model the time series; such a model performs better than a model with the same lag but more or fewer (e.g. 50 or 200) hidden units.

Lag 50 & 100 hidden units give the best performance (MSE 418.086) on the test set. Lag 10 & 100 hidden units give second-best performance (MSE 1643.964) on the test set. Further experiments might reveal an even better configuration, e.g. $10 \leq \text{lag} \leq 50$.

2.2.3 Model Evaluation

The MLP requires a much larger number of inputs (i.e. lag) in order to be able to learn the underlying trend in the time series, and even so performs worse especially when it comes to predicting unexpected events. The LSTM requires fewer inputs due to its more sophisticated architecture, and is better able to predict unexpected events. The LSTM clearly outperforms in terms of performance and in terms of its requirements for past data availability in order to be able to make predictions.

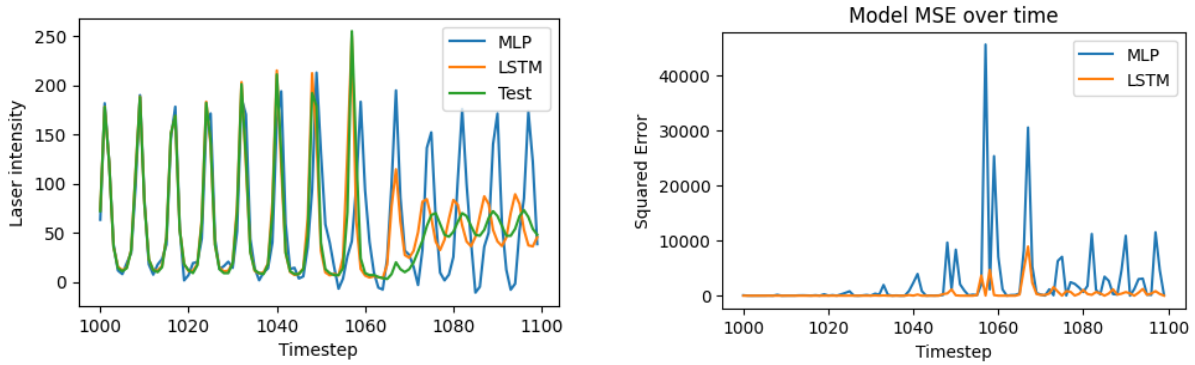

Figure 10: Predictions on test set (MLP MSE: 2462.667, LSTM MSE: 418.086)

3 Deep Feature Learning

3.1 Image Reconstruction with Autoencoders

An autoencoder performs nonlinear dimensionality reduction by minimising the *reconstruction* error, subject to a *sparsity* constraint in the number of hidden units - enforcing learning not the identity which would be trivial but a lower-dimensional latent structure.

The sparser the architecture, the more difficult it is to learn a meaningful representation (as characterised by the reconstruction loss, as optimisation metric). ≥ 32 hidden units trained for ≥ 10 epochs are sufficient to give good reconstructed images.

Train/Test Losses	16	32	64	128 Hidden Units
Epoch 1	2.79 / 1.93	2.33 / 1.44	1.99 / 1.20	1.69 / 0.93
Epoch 10	0.98 / 0.94	0.52 / 0.49	0.27 / 0.25	0.13 / 0.13
Epoch 20	0.91 / 0.88	0.45 / 0.43	0.20 / 0.20	0.08 / 0.09
Epoch 40	0.88 / 0.85	0.43 / 0.42	0.18 / 0.18	-
Epoch 80	0.85 / 0.83	-	-	-

Table 1: Results for training an autoencoder on the MNIST dataset ($28 \times 28 = 784$ dims)

3.2 Image Classification with Stacked Autoencoders

Pre-Training A stacked autoencoder is (pre-)trained by greedy layer-wise training on successive layers of autoencoders. The encoder plus a FC layer can be used as a classifier.

Pre-training has the *benefits* of learning features independently of the final task in an unsupervised fashion (e.g. there may not be enough annotated data available for supervised learning), and layer-by-layer which avoids vanishing gradients that can occur if training deep networks (e.g. with potentially badly initialised weights) from scratch.

Network Architecture 256 & 64 hidden units in the first & second layer, respectively, give the highest test set accuracy. A third layer might improve performance for complex features. Generally fine-tuning is used to improve performance on a supervised task.

Test accuracy	(64, 32)	(128, 32)	(128, 64)	(256, 64)	(256, 128)	(512, 128)
Pre-trained	0.104	0.108	0.098	0.215	0.108	0.163
Fine-tuned	0.929	0.914	0.930	0.916	0.914	0.900

Table 2: Accuracy on the test set for different number of hidden units in layers 1 and 2

Fine-Tuning In greedy layer-wise training, the parameters of each layer are trained individually (with the other layers' parameters being frozen in the meantime). Fine-tuning runs backpropagation end-to-end on the whole model to update and improve ("fine-tune") the (pre-trained) parameters across all layers, for a specific *supervised* task.

Fine-tuning improves performance on the classification task across the board for all architectures in the experiment. Note that the network that achieved the best test metric pre-trained does not perform best following fine-tuning. But in general, with fine-tuning, test accuracy is very high across the board, and networks with fewer hidden units are able to achieve very good test accuracy. In particular, choosing double the number of hidden units in the first relative to the second layer achieves the best results in the experiment.

3.3 Convolutional Neural Networks (CNNs)

3.3.1 Convolutions

The dimensionality of the output of a convolutional layer is:

$$\left\lfloor \frac{n_h - k_h + p_h + s_h}{s_h} \right\rfloor \times \left\lfloor \frac{n_w - k_w + p_w + s_w}{s_w} \right\rfloor \qquad [31] \qquad (6)$$

For example: $\dim\left(X_{4\times4} *_{s=2}^{p=0} K_{2\times2}\right) = \lfloor(4-2+0+2)/2\rfloor \times \lfloor(4-2+0+2)/2\rfloor = 2\times2.$

$$\begin{bmatrix} 2 & 5 & 4 & 1 \\ 3 & 1 & 2 & 0 \\ 4 & 5 & 7 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix} *_{s=2}^{p=0} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \sum \left(\begin{bmatrix} 2 & 5 \\ 3 & 1 \end{bmatrix} \odot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \sum \left(\begin{bmatrix} 4 & 1 \\ 2 & 0 \end{bmatrix} \odot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \\ \sum \left(\begin{bmatrix} 4 & 5 \\ 1 & 2 \end{bmatrix} \odot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \sum \left(\begin{bmatrix} 7 & 1 \\ 3 & 4 \end{bmatrix} \odot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 6 & 11 \end{bmatrix} \qquad (7)$$

CNNs differ from fully connected (FC) networks: [9]

1. *Local connectivity / sparsity of connections*: neurons closer to each other are likely more related than ones farther away due to locality in images - thus, in each layer, each output value depends on a much smaller number of inputs, as opposed to FC.
2. *Parameter sharing*: a feature detector (e.g. a vertical edge detector) that is useful in one part of the image is probably useful in another part of the image [17] - sharing weights across the image further reduces the number of parameters relative to FC.

Local connectivity and parameter sharing enable CNNs to achieve good generalisation even with less training data and to be able to process larger images.

Parameter sharing and pooling[8] enable *translation and shift invariance*, $f(t(x)) = f(x)$, i.e. $(f \circ t) = f$, s.t. features can be detected in any position in an image.

Moreover, while deep neural networks in general leverage *(hierarchical) feature learning*[9], CNN's architectural properties make them particularly effective for images.

3.3.2 Digit Recognition with Convolutional Neural Networks (CNNs)

An incremental training strategy [8] is applied: first the number of convolutional layers and filters is tuned, the kernel size, and then the number of fully connected layers.

The number of convolutional layers in a CNN determines the depth of hierarchical feature extraction, allowing the network to learn increasingly abstract representations of the input. Each layer captures progressively higher-level features from the image. The number of filters in each convolutional layer determines how many distinct features, such as edges or higher-dimensional shapes, the layer can detect in parallel.

Digits consist of a small number of similar patterns like loops, vertical, horizontal and diagonal lines. Two or three conv layers with a suitable number of filters should suffice.

A model architecture with three conv, one FC layer, kernel size 3 is selected.

[8]The second main component in CNNs is pooling [6], e.g. avg (linear operation), min/max (nonlinear). A pooling layer detects features regardless of their position in the image by trading off spatial resolution.

[9]While classical methods, e.g. for edge detection, require features to be user-defined, such as *fixed* convolution kernels, neural networks learn features themselves directly from the data and hierarchically, i.e. different layers learning features of different granularity such as edges, etc.

3.4 Attention and Transformers

3.4.1 The Self-Attention Mechanism

In cognitive science, attention is defined as the mechanism of selectively focusing on a particular aspect of information that is of interest [7]. In information retrieval, natural language processing and other AI applications, attention is a neural network architecture designed to retrieve a weighted combination of values given a query's similarity to the value's key, such as by (scaled) dot product, $\frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle}{\sqrt{d}}$, or cosine similarity, $\frac{\langle \mathbf{q}_i, \mathbf{k}_j \rangle}{\|\mathbf{q}_i\| \|\mathbf{k}_j\|}$. [20, 28]

$$\text{Attention}(q_i, \mathbf{k}, \mathbf{v}) = \sum_{j=1}^{n} \text{Similarity}(q_i, k_j) \times v_j \tag{8}$$

This addresses a key limitation of *static embeddings*, e.g. GloVe, Word2Vec, which assign a fixed representation to words regardless of context. Attention enables *dynamic embeddings*, e.g. BERT, GPT, which incorporate contextual information by computing each token's representation as a weighted combination of all sequence tokens. This allows polysemous words like "bank" to receive different representations depending on context.

Interpretation of Q, K, V matrices The *key*, k_j, is an index of what information position j contains. The *value*, v_j, contains the actual information content of position j in the output. Similarity between queries, q_i, and keys, k_j, determines how much the value corresponding to the respective key should contribute to the output for each query.

In *self-attention*, the query and the key are both from the same sequence. That is, self-attention computes a representation of the same sequence or input by determining which parts of the sequence should be focused on, i.e. 'attended to'.[10] The self-attention mechanism as defined in the original paper [26] applies softmax[11] as similarity and normalisation to ensure attention weights sum to 1 and are non-negative.[12]

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{9}$$

Dimensional relationship between attention scores and outputs Given input sequence length n and the model's embedding dimension d, inputs X are $n \times d$, and projection weights W_q, W_k, W_v are $d \times d$ s.t. X projected onto the weights gives queries matrix $Q = XW_q$, keys matrix $K = XW_k$, values matrix $V = XW_v$ all of dimensions $n \times d$, where the rows each represent a query / key / value vector for a single input.

Every query is compared to every key, i.e. the (scaled) dot product QK^T is $n \times$n, s.t. the *attention scores* have shape $n \times n$ - unchanged in dimension by softmax normalisation.

Finally, the *attention outputs* are obtained from the dot product of the attention scores and the values matrix V and therefore have shape $(n \times n) \times (n \times d) = n \times d$. Each output row corresponds to an input and is a weighted sum of values by attention scores.

[10] Other forms of attention include *cross(-domain)* or encoder-decoder attention where the query comes from the *decoder* while the key and value are from the *encoder* as used in sequence-to-sequence tasks like machine translation, and *cross-modal* attention where the query, key and value are from different *modalities* (e.g. word embeddings as a query for images) as used in multimodal transformers like CLIP. Moreover, *masked* self-attention involves masking to prevent later tokens from influencing earlier tokens in order to ensure that the output at each step only attends to already-seen parts of the sequence. [20, 28]

[11] Temperature scaling can be used to control the sharpness of the softmax probability distribution: higher temperatures produce a more uniform, lower temperatures a more concentrated distribution.

[12] Further, it is important to note that *positional encodings* are needed to distinguish between sequences with identical tokens in a different order, since attention is *permutation-equivariant* to the inputs. [20]

Query/key dimension d_q and value dimension d_v can differ: $d_v \neq d_q$ decouples similarity computation (on queries, keys) from information aggregation (with values). [20]

3.4.2 Vision Transformers (ViT)

The Transformer architecture [26] leverages the self-attention mechanism along with a residual connection, layer normalisation, feedforward neural network (FFNN), and multiple encoder blocks. A major improvement over RNNs is that transformers are able to process the entire sequence in parallel. Moreover, while CNNs and RNNs are modality-specific, the attention mechanism can in principle be applied to any modality[13]. [20, 28]

The Vision Transformer (ViT) [4] creates patch embeddings from an image by creating a grid of 16×16 patches. These are projected through a learnable linear transformation that flattens a 2D image into a D-dimensional vector, a positional encoding is added, and the embeddings are passed into the encoder as inputs. The architecture consists of multi-head self-attention layers and multi-layer perceptrons which are alternated. A class token, `<cls>`, gives the predictions of the class probabilities. [20, 28]

Network Size and Hyperparameter Tuning The default baseline model [9] uses an embedding dimension of 64, 6 layers, 8 heads, a MLP dimension of 128. Increasing the *embedding dimension* improves train and eval predictions. Comparing *network depth* shows that deeper networks perform better but with increased computational cost, training for longer. Reducing *attention heads* improves generalisation performance, indicating that not that many heads are needed when the dataset is not that large. The small to modest *feedforward network dimension* has the best effect on out-of-sample performance.

Architecture				Dropout	Epochs	Train (Loss / Acc)	Eval (Loss / Acc)
64	6	8	128	0.0	15	0.0699 / 99.07%	0.0583 / 98.12%
32	6	8	128	0.0	23	0.0323 / 99.16%	0.0712 / 98.00%
128	6	8	128	0.0	18	0.0003 / 99.57%	0.0516 / 98.59%
64	**4**	8	128	0.0	15	0.0182 / 99.15%	0.0591 / 98.15%
64	**8**	8	128	0.0	22	0.0349 / 99.61%	0.0562 / 98.61%
64	6	**4**	128	0.0	18	0.0129 / 99.49%	0.0563 / 98.31%
64	6	8	**64**	0.0	14	0.0291 / 98.94%	0.0636 / 98.11%
64	6	8	**256**	0.0	11	0.0565 / 98.20%	0.0596 / 98.03%
64	6	8	128	**0.1**	15	0.0117 / 98.64%	0.0513 / 98.43%
64	6	8	128	**0.2**	20	0.0351 / 98.90%	0.0512 / 98.49%

Table 3: Vision Transformer (ViT) hyperparameter tuning results

Regularisation Early stopping with patience 5 serves as (implicit) regularisation (and saves resources). It can "be used ... alone or in conjunction with other ... strategies". [9]

Further, dropout[14] is a commonly used regularisation technique that randomly sets a fraction of input units to zero during training, preventing the model from becoming overly dependent on specific features. Given ViT's large parameter counts and potential for memorising training data, 10-20% dropout is beneficial for improved generalisation.

[13] Creating the embeddings for different modalities for Transformers is modality-specific. There is also a limit to the granularity with which they can be created, due to quadratic scaling of self-attention. Tokenisation in NLP produces embeddings that are relatively granular while still being scalable. [20, 28]

[14] "Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble" [9].

4 Generative Models

4.1 Energy-Based Models

4.1.1 Restricted Boltzmann Machines (RBMs)

Pseudolikelihood Computing the log likelihood for MLE on a RBM requires summing over $2^{|v'|+|h'|}$ combinations of binary visible and hidden variables $v' \in \{0,1\}, h' \in \{0,1\}$.

$$\theta^*_{\text{MLE}} = \arg\max_\theta P(v \mid \theta) = \arg\max_\theta \sum_h P(v,h \mid \theta) = \arg\max_\theta \mathcal{L}(\theta) \propto \arg\max_\theta \log \mathcal{L}(\theta) \tag{10}$$

$$\log \mathcal{L}(\theta) = \log \frac{1}{Z(\theta)} \sum_h e^{-E(v,h;\theta)} = \log \sum_h e^{-E(v,h;\theta)} - \log \sum_{v',h'} e^{-E(v',h';\theta)} \tag{11}$$

The (log) pseudolikelihood [9] approximates the (log) likelihood as the product of visible states v_i''s conditional probabilities given all other visible states $v_{j\neq i}$.

$$\log \mathcal{L}(\theta) = \log P(v \mid \theta) = \log \prod_i P(v_i \mid v_{j\neq i}; \theta) = \sum_i \log P(v_i \mid v_{j\neq i}; \theta) \tag{12}$$

Pseudolikelihood (PL) is an *approximation* technique that makes the training faster and *tractable* on larger networks. Having said that, training based on an approximation technique will not be as precise as an exact method. The quality of learned representations will depend on the extent to which PL is able to approximate the true likelihood.

Another popular approximation used for training RBMs is Contrastive Divergence. CD uses sampling (namely Gibbs sampling), and as such is *stochastic*; PL is *deterministic*.

Hyperparameter Tuning Increasing the number of *components* or hidden units improves performance. With 100 components it is possible to learn more patterns than with 50 or 10. But there are signs of diminishing returns at this point. Moreover, it becomes computationally heavier and much slower (despite calculating the *pseudo*likelihood).

A *learning rate* of 0.01 is appropriate for 100 components. For 10, ≈ 0.1 is better.[15]

Increasing the number of *iterations* from 10 to 20 or 30 yields marginally better pseudolikelihood values. Given that the training also takes 2 or 3 times as long, more than 20 iterations may not be worth it unless maximising pseudolikelihood is paramount.

No. of components	Learning rate	No. of iterations	PL	Avg. time / it.
10	0.01	10	-187.61	4.1s
50	0.01	10	-98.75	5.9s
100	0.01	10	-79.74	13.2s
100	0.001	10	-109.78	13.7s
100	0.05	10	-87.49	13.6s
100	0.1	10	-96.93	13.5s
100	0.01	20	-76.54	13.2s
100	0.01	30	-75.07	13.2s

Table 4: Training a Bernoulli RBM on MNIST with different hyperparameters

[15] As in general, the trade-off is between training speed, risk of stagnation or overshooting minima.

A model with more hidden units can learn a more varied set of latent features. A model with fewer hidden units learns fewer, more general features. In unsupervised learning there is no overfitting but since a very large model has a lot of capacity it could memorise data and may have reduced ability to generate diverse or novel samples. A good model should produce a distribution that manages to capture the training distribution and to reliably generate similar outputs, without explicitly memorising training inputs.

Gibbs Sampling from Model Gibbs sampling from the trained model with test data demonstrates how it can generate variations of the test data based on the learned data distribution. It is a Markov Chain Monte Carlo (MCMC) technique to draw from the joint distribution by alternating between $h \sim P(h \mid v)$ and $v \sim P(v \mid h)$ s.t. after some number of steps the sample should resemble the learned model distribution, $v \sim P(v)$.

Taking only one Gibbs sampling step generates data that is minimally different from the test data. The more Gibbs sampling steps are performed, the more representative the generated images become of the distribution of representations learned by the model.

Image Reconstruction Reconstructions are not perfect but with 1 to 5 Gibbs steps the model is able to reconstruct images with up to 15 rows quite well. With more Gibbs steps, e.g. 100, the reconstruction for close to half of the digits diverges and looks quite different; this is to be expected. With 20 or more rows removed in the original image, the digit in it is not recognisable anymore, and the model is not able to reconstruct it. A 2, e.g., is most difficult to reconstruct if the lower rows of the image are removed.

4.1.2 Deep Boltzmann Machines (DBMs)

Filters / Interconnection Weights A Deep Boltzmann Machine (DBM) consists of multiple layers of hidden units, where each layer is connected only to the adjacent layers (i.e., no intra-layer or skip connections). The features learned in the first hidden layer are simple patterns like edges or shapes, similar to those learned in the RBM. The second hidden layer captures more complex combinations of features from the first layer. Multiple layers enable learning more complex, hierarchical representations.

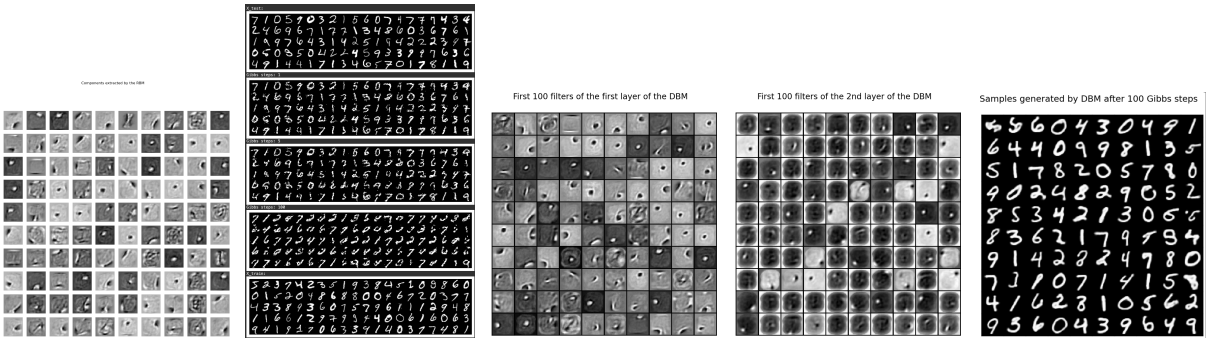

Figure 11: RBM vs. DBM comparison: learned features and sampled images

Model Evaluation There are still some outputs that do not look like a digit. But overall the quality of the samples generated by the DBM is higher and they are more realistic. The distribution is more random and a greater variety of different digits is generated. The multiple layers enable the DBM to learn more high-level features and therefore to generalise better in terms of generation and reconstruction of images.

4.2 Generative Adversarial Networks (GANs)

4.2.1 Loss Functions

Generative Adversarial Networks (GANs) are set up as a zero-sum game between a Generator network (G) which generates samples and receives payoff $-v(\theta^{(G)}, \theta^{(D)})$ and a Discriminator network (D) which attempts to distinguish between training and generated samples and receives payoff $v(\theta^{(G)}, \theta^{(D)})$. The Generator aims to minimise this value, while the Discriminator aims to maximise it, i.e. at convergence $G^* = \arg\min_G \max_D v(G, D)$.

4.2.2 Discriminator vs. Generator Performance

If the Discriminator performs proportionally much better than the Generator, it means that $D(x)$ is very accurate in distinguishing between real and fake samples, and its outputs for fake / real samples, $D(G(z))$, are consistently low (close to 0) / high (close to 1). This limits the Generator's ability to learn because it receives little feedback how to improve.[16]

4.2.3 Convergence and Stability

The objective is to find a Nash equilibrium, a saddle point, where neither player can improve their payoff by unilaterally changing their strategy. At convergence, the Generator's distribution p_G should be similar to the data distribution p_{data}, and the Discriminator should be unable to differentiate between the two, i.e. $D(x) \approx 1/2$ for all inputs. However, the adversaries might not perform proportionally and the objective $\max_D v(G, D)$ is often not convex, which makes the optimisation problem challenging. In the PyTorch implementation, momentum is reduced by decreasing the exponential decay applied to the gradient, in order to thus enable better player reactions and stable training progress.

4.2.4 Latent Space

Data is generated by sampling from the latent space, a lower-dimensional representation. Interpolating the latent space shows a realistic three but at $\lambda = 1.0$, a different unclear digit appears. That is, there is some smooth transition in the latent space but digit identity does not appear disentangled from other latent features or noise, esp. at $\lambda = 1.0$.

4.2.5 CNN Backbone

Integration of CNNs into the Generator and Discriminator architectures, e.g. as in the Deep Convolutional Generative Adversarial Network (DCGAN) architecture, gives images with better spatial coherence and improves results. Training stability is similarly difficult.

4.2.6 Advantages and Disadvantages

A key advantage of GANs is that their learning process, which is set up as an adversarial problem, does not require approximate inference or approximation of a partition function. Challenges include non-convexity of the objective, vanishing gradients (e.g. when the discriminator becomes too strong), and difficulty of converging to a stable equilibrium (saddle points). VAEs, on the other hand, are generally more stable to train because they optimise a well-defined lower bound (ELBO; through variational inference).

The samples generated by the GAN model are higher in quality than by the VAE.

[16] Especially in the case of a traditional minimax GAN, where the Generator minimises $E_{z\sim p_z} \log(1 - D(G(z)))$, if $D(G(z))$ is very close to 0 (i.e. the Discriminator confidently identifies generated samples as fake), the term $\log(1 - D(G(z)))$ would be close to $\log(1)$, which is 0, causing *vanishing gradients*, as the gradient of the loss w.r.t. the Generator's parameters would become very small. Non-saturating GANs, $-E_{z\sim p_z} \log(D(G(z)))$, address this issue; still, a stronger Discriminator would impede learning.

4.3 Variational Autoencoders (VAEs)

4.3.1 Optimisation Metric

The Variational Autoencoder (VAE) model does not maximise the log-likelihood $\ln p(x)$ directly, but rather it maximises a lower bound on this log-likelihood called the Evidence Lower BOund (ELBO). This is because direct computation of $\ln p(x) = \ln \int p(x|z)p(z)dz$ is often intractable due to the integral over the latent variable z.

The ELBO is derived by applying Jensen's inequality to the log-likelihood. It can be expressed as: ELBO $= E_{z\sim q_\phi(z|x)}[\ln p(x|z)] - E_{z\sim q_\phi(z|x)}[\ln q_\phi(z|x) - \ln p(z)]$. This lower bound is then maximised during training. The first part is referred to as the (negative) reconstruction error. It encourages the decoder $p_\theta(x|z)$ to accurately reconstruct the input data x from its latent representation z sampled from the approximate posterior $q_\phi(z|x)$. The second part is the Kullback-Leibler (KL) divergence, $\mathrm{KL}[q_\phi(z|x)\|p(z)]$, which encourages the approximate posterior $q_\phi(z|x)$ to stay close to a prior, e.g. $p(z) \sim \mathcal{N}$.

4.3.2 Reconstruction Error

(Stacked) autoencoders minimise reconstruction error in the form of *squared distortion error*, which is essentially the MSE, $\min E = \frac{1}{N}\sum_{i=1}^{N} \|x_i - F(G(x_i))\|_2^2$. Variational autoencoders are a probabilistic model maximising ELBO, i.e. reconstruction error - KL divergence. The reconstruction error for VAEs is the *negative log-likelihood*, $-\ln p(x|z)$.[17]

4.3.3 Latent Space

VAEs are good at creating well-behaved latent spaces[18], where every latent variable z corresponds to a plausible data example x. Interpolating two latent vectors (corresponding to digits) with a value from 0.0 to 1.0, shows how the images transition smoothly from one to the other. The KL divergence acts as a regularising term encouraging such a structured latent space following a prior distribution.

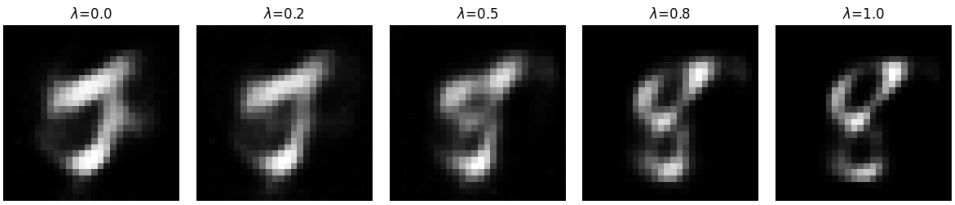

Figure 12: Interpolation in VAE latent space

4.3.4 Generation Mechanism

In GANs, the Generator produces data by mapping a random noise vector z (sampled from a simple prior distribution) to a data sample $x = G(z; \theta^{(G)})$. This process is feed-forward and data is synthesised without reference to existing inputs.

In VAEs, generation also starts by sampling $z \sim p(z)$. The decoder $p_\theta(x \mid z)$ then generates a data sample x from the latent variable. Training also requires the encoder, but generation also relies solely on the decoder and prior.

[17]The negative log-likelihood depends on the output distribution, which e.g. for Gaussians is also the MSE, or binary cross-entropy (BCE) for Bernoulli-distributed data (i.e. binary data).

[18]Another desirable property is a disentangled latent space, where manipulating each dimension of z corresponds to changing an interpretable property of the data.

A Delta Rule

Consider the outputs $x^L_{i,p}$ of a neural network, and the mean-squared error (MSE) empirical risk[19] E which is to be minimised for the patterns (i.e. data points) $p = 1, \dots, P$:

$$E = \frac{1}{P}\sum_{p=1}^{P} E_p = \frac{1}{2P}\sum_{p=1}^{P}\sum_{i=1}^{N_L}(x^L_{i,p} - x^{\text{desired}}_{i,p})^2 \tag{13}$$

$$\begin{aligned}
x^L_{i,p} &= \sigma(\xi^L_{i,p}), \qquad & \xi^L_{i,p} &= \sum_{j=1}^{N_{L-1}} w^L_{i,j} x^{L-1}_{j,p}, \\
x^{L-1}_{j,p} &= \sigma(\xi^{L-1}_{j,p}), \qquad & \xi^{L-1}_{j,p} &= \sum_{k=1}^{N_{L-2}} w^{L-1}_{j,k} x^{L-2}_{k,p}, \\
& & &\vdots \\
x^2_{r,p} &= \sigma(\xi^2_{r,p}), \qquad & \xi^2_{r,p} &= \sum_{s=1}^{N_1} w^2_{r,s} x^1_{s,p}, \\
x^1_{s,p} &= \sigma(\xi^1_{s,p}), \qquad & \xi^1_{s,p} &= \sum_{t=1}^{N_0} w^1_{s,t} x^0_{t,p}
\end{aligned} \tag{14}$$

The empirical risk can be minimised by iteratively updating the weights:

$$\Delta w^l_{i,j} \leftarrow -\eta \min_{w^l_{i,j}} E \quad \equiv \quad \Delta w^l_{i,j} \leftarrow -\eta \frac{\partial E}{\partial w^l_{i,j}} \quad \forall i, j; l = 1, \dots, L \tag{15}$$

These partial derivatives can be computed by applying the *chain rule*:

$$\begin{aligned}
\frac{\partial E}{\partial w^L_{i,j}} &= \frac{\partial}{\partial w^L_{i,j}} \frac{1}{2P} \sum_{p=1}^{P}\sum_{i=1}^{N_L}(x^L_{i,p} - x^{\text{desired}}_{i,p})^2 = \frac{1}{P}\sum_{p=1}^{P}\sum_{i=1}^{N_L}(x^L_{i,p} - x^{\text{desired}}_{i,p}) \frac{\partial}{\partial w^L_{i,j}} x^L_{i,p} \\
&= \frac{1}{P}\sum_{p=1}^{P}\sum_{i=1}^{N_L}(x^L_{i,p} - x^{\text{desired}}_{i,p}) \sigma'(\xi^L_{i,p}) \frac{\partial}{\partial w^L_{i,j}} \xi^L_{i,p} \\
&= \frac{1}{P}\sum_{p=1}^{P}\sum_{i=1}^{N_L} \delta^L_{i,p} x^{L-1}_{j,p}, \qquad \text{where} \quad \delta^L_{i,p} = (x^L_{i,p} - x^{\text{desired}}_{i,p})\sigma'(\xi^L_{i,p}) \\
&= \frac{1}{P}\sum_{p=1}^{P} \delta^L_{i,p} x^{L-1}_{j,p}, \qquad \text{if there is a single output neuron, i.e. } N_L = 1
\end{aligned} \tag{16}$$

[19] The *error* is the difference between predicted and desired output, $x^L_{j,p} - x^d_{j,p}$, for a pattern p. *Loss* is a measure of the prediction quality, e.g. the mean squared error (MSE), $E_p = \frac{1}{2}\sum_{j=1}^{N_L}(x^L_{j,p} - x^d_{j,p})^2$, for a pattern p. *Risk* is the expected loss over the entire data distribution. *Empirical risk* or *training error* is the average loss over the training dataset or expected empirical loss, $E = \frac{1}{P}\sum_{p=1}^{P} E_p$. [27]

$$
\begin{aligned}
\frac{\partial E}{\partial w_{j,k}^{L-1}} &= \frac{\partial}{\partial w_{j,k}^{L-1}} \frac{1}{2P} \sum_{p=1}^{P} \sum_{i=1}^{N_L} (x_{i,p}^{L} - x_{i,p}^{\text{desired}})^2 = \frac{1}{P} \sum_{p=1}^{P} \sum_{i=1}^{N_L} \delta_{i,p}^{L} w_{i,j}^{L} \frac{\partial}{\partial w_{j,k}^{L-1}} x_{j,p}^{L-1} \\
&= \frac{1}{P} \sum_{p=1}^{P} \sum_{i=1}^{N_L} \delta_{i,p}^{L} w_{i,j}^{L} \sigma'(\xi_{j,p}^{L-1}) \frac{\partial}{\partial w_{j,k}^{L-1}} \xi_{j,p}^{L-1} = \frac{1}{P} \sum_{p=1}^{P} \sum_{i=1}^{N_L} \delta_{i,p}^{L} w_{i,j}^{L} \sigma'(\xi_{j,p}^{L-1}) x_{k,p}^{L-2} \\
&= \frac{1}{P} \sum_{p=1}^{P} \delta_{j,p}^{L-1} x_{k,p}^{L-2}, \qquad \text{where} \quad \delta_{j,p}^{L-1} = \Big(\sum_{i=1}^{N_L} \delta_{i,p}^{L} w_{i,j}^{L} \Big) \sigma'(\xi_{j,p}^{L-1})
\end{aligned}
\tag{17}
$$

$$
\vdots
$$

$$
\begin{aligned}
\frac{\partial E}{\partial w_{s,t}^{1}} &= \frac{\partial}{\partial w_{s,t}^{1}} \frac{1}{2P} \sum_{p=1}^{P} \sum_{i=1}^{N_L} (x_{i,p}^{L} - x_{i,p}^{\text{desired}})^2 = \frac{1}{P} \sum_{p=1}^{P} \sum_{i=1}^{N_L} \delta_{i,p}^{L} \sum_{j=1}^{N_{L-1}} w_{i,j}^{L} \frac{\partial}{\partial w_{s,t}^{1}} x_{j,p}^{L-1} \\
&= \frac{1}{P} \sum_{p=1}^{P} \sum_{i=1}^{N_L} \delta_{i,p}^{L} \sum_{j=1}^{N_{L-1}} w_{i,j}^{L} \sigma'(\xi_{j,p}^{L-1}) \frac{\partial}{\partial w_{s,t}^{1}} \xi_{j,p}^{L-1} \\
&= \frac{1}{P} \sum_{p=1}^{P} \sum_{j=1}^{N_{L-1}} \delta_{j,p}^{L-1} \sum_{k=1}^{N_{L-2}} w_{j,k}^{L-1} \frac{\partial}{\partial w_{s,t}^{1}} x_{k,p}^{L-2}, \quad \text{where} \quad \delta_{j,p}^{L-1} = \Big(\sum_{i=1}^{N_L} \delta_{i,p}^{L} w_{i,j}^{L} \Big) \sigma'(\xi_{j,p}^{L-1}) \\
&= \frac{1}{P} \sum_{p=1}^{P} \sum_{j=1}^{N_{L-1}} \delta_{j,p}^{L-1} \sum_{k=1}^{N_{L-2}} w_{j,k}^{L-1} \sigma'(\xi_{k,p}^{L-2}) \frac{\partial}{\partial w_{s,t}^{1}} \xi_{k,p}^{L-2} \\
&= \frac{1}{P} \sum_{p=1}^{P} \sum_{k=1}^{N_{L-2}} \delta_{k,p}^{L-2} \frac{\partial}{\partial w_{s,t}^{1}} \xi_{k,p}^{L-2}, \qquad \text{where} \quad \delta_{k,p}^{L-2} = \Big(\sum_{j=1}^{N_{L-1}} \delta_{j,p}^{L-1} w_{j,k}^{L-1} \Big) \sigma'(\xi_{k,p}^{L-2}) \\
&\vdots \\
&= \frac{1}{P} \sum_{p=1}^{P} \sum_{r=1}^{N_2} \delta_{r,p}^{2} w_{r,s}^{2} \sigma'(\xi_{s,p}^{1}) \frac{\partial}{\partial w_{s,t}^{1}} \xi_{s,p}^{1} \\
&= \frac{1}{P} \sum_{p=1}^{P} \delta_{s,p}^{1} x_{t,p}^{0}, \qquad \text{where} \quad \delta_{s,p}^{1} = \Big(\sum_{r=1}^{N_2} \delta_{r,p}^{2} w_{r,s}^{2} \Big) \sigma'(\xi_{s,p}^{1})
\end{aligned}
\tag{18}
$$

The generalised delta rule[20] captures the recursive nature of the derivatives:

$$
\begin{aligned}
\Delta w_{i,j}^{l} \leftarrow -\eta \frac{\partial E}{\partial w_{i,j}^{l}} &= -\eta \frac{1}{P} \sum_{p=1}^{P} \frac{\partial E}{\partial \xi_{i,p}^{l}} \frac{\partial \xi_{i,p}^{l}}{\partial w_{i,j}^{l}} = -\eta \frac{1}{P} \sum_{p=1}^{P} \delta_{i,p}^{l} x_{j,p}^{l-1}, \quad l = 1, \ldots, L \\
\delta_{i,p}^{L} &= \frac{\partial E}{\partial \xi_{i,p}^{L}} = (x_{i,p}^{L} - x_{i,p}^{\text{desired}}) \sigma'(\xi_{i,p}^{L}), \quad \text{for output layer with } N_L = 1 \\
\delta_{j,p}^{l} &= \frac{\partial E}{\partial \xi_{j,p}^{l}} = \Big(\sum_{i=1}^{N_{l+1}} \delta_{i,p}^{l+1} w_{i,j}^{l+1} \Big) \sigma'(\xi_{j,p}^{l}), \quad l = 1, \ldots, L-1
\end{aligned}
\tag{19}
$$

[20] Mini-batch or stochastic gradient descent (SGD) considers one pattern at a time, i.e. $P = 1$. The overall effect of averaging over the patterns then applies over batches or iterations.

B Backprop Example

Activation function $\sigma(t) = \frac{1}{1+\exp(-t)}$, $\sigma'(t) = \frac{d\sigma}{dt} = \sigma(t)(1-\sigma(t))$:

$$\begin{aligned}\frac{d\sigma}{dt} &= -(1+\exp(-t))^{-2}\exp(-t)(-1) = \frac{\exp(-t)}{\big(1+\exp(-t)\big)^2} = \sigma(t)\frac{\exp(-t)}{1+\exp(-t)} \\ &= \sigma(t)\frac{1+\exp(-t)-1}{1+\exp(-t)} = \sigma(t)\left(\frac{1+\exp(-t)}{1+\exp(-t)} - \frac{1}{1+\exp(-t)}\right) = \sigma(t)\big(1-\sigma(t)\big)\end{aligned} \tag{20}$$

Layer $l=0$	Data point $p=1$	Data point $p=2$	Data point $p=3$
Component $i=1$ (bias)	$x^0_{1,1}=1.0$	$x^0_{1,2}=1.0$	$x^0_{1,3}=1.0$
Component $i=2$	$x^0_{2,1}=1.0$	$x^0_{2,2}=0.0$	$x^0_{2,3}=-1.0$
Component $i=3$	$x^0_{3,1}=1.0$	$x^0_{3,2}=-2.0$	$x^0_{3,3}=1.0$

Table 5: Layer $l=0$: inputs $x^l_{i,p}$

Layer $l=1$	Source $j=1$ (bias)	Source $j=2$	Source $j=3$
Component $i=2$	$w^1_{2,1}=0.4$	$w^1_{2,2}=0.2$	$w^1_{2,3}=-1.2$
Component $i=3$	$w^1_{3,1}=0.7$	$w^1_{3,2}=-0.7$	$w^1_{3,3}=-0.7$

Table 6: Layer $l=1$: weights $w^l_{i,j}$

Layer $l=1$	Data point $p=1$	Data point $p=2$	Data point $p=3$
Component $i=2$	$\xi^1_{2,1}=\sum_{j=1}^3 w^1_{2,j}x^0_{j,1} = [0.4,0.2,-1.2]\cdot[1,1,1]=-0.6$	$\xi^1_{2,2}=\sum_{j=1}^3 w^1_{2,j}x^0_{j,2} = [0.4,0.2,-1.2]\cdot[1,0,-2]=2.8$	$\xi^1_{2,3}=\sum_{j=1}^3 w^1_{2,j}x^0_{j,3} = [0.4,0.2,-1.2]\cdot[1,-1,1]=-1.0$
Component $i=3$	$\xi^1_{3,1}=\sum_{j=1}^3 w^1_{3,j}x^0_{j,1} = [0.7,-0.7,-0.7]\cdot[1,1,1]=-0.7$	$\xi^1_{3,2}=\sum_{j=1}^3 w^1_{3,j}x^0_{j,2} = [0.7,-0.7,-0.7]\cdot[1,0,-2]=2.1$	$\xi^1_{3,3}=\sum_{j=1}^3 w^1_{3,j}x^0_{j,3} = [0.7,-0.7,-0.7]\cdot[1,-1,1]=0.7$

Table 7: Layer $l=1$: interconnections with previous layer $\xi^l_{i,p}=\sum_{j=1}^{N_{l-1}} w^l_{i,j}x^{l-1}_{j,p}$, $N_0=3$

Layer $l=1$	Data point $p=1$	Data point $p=2$	Data point $p=3$
Component $i=1$ (bias)	$x^1_{1,1}=1.0$	$x^1_{1,2}=1.0$	$x^1_{1,3}=1.0$
Component $i=2$	$x^1_{2,1}=\sigma(\xi^1_{2,1})=0.354$	$x^1_{2,2}=\sigma(\xi^1_{2,2})=0.942$	$x^1_{2,3}=\sigma(\xi^1_{2,3})=0.269$
Component $i=3$	$x^1_{3,1}=\sigma(\xi^1_{3,1})=0.332$	$x^1_{3,2}=\sigma(\xi^1_{3,2})=0.891$	$x^1_{3,3}=\sigma(\xi^1_{3,3})=0.668$

Table 8: Layer $l=1$: neurons $x^l_{i,p}=\sigma(\xi^l_{i,p})$

Layer $L=2$	Source $j=1$ (bias)	Source $j=2$	Source $j=3$
Component $i=1$	$w^2_{1,1}=0.4$	$w^2_{1,2}=-0.5$	$w^2_{1,3}=2.3$

Table 9: Layer $L=2$: weights $w^l_{i,j}$

Layer $L=2$	Data point $p=1$	Data point $p=2$	Data point $p=3$
Component $i=2$	$\xi^2_{1,1}=\sum_{j=1}^3 w^2_{1,j}x^1_{j,1}$ $=[0.4,-0.5,2.3]\cdot$ $[1.0,0.354,0.332]=$ 0.986	$\xi^2_{1,2}=\sum_{j=1}^3 w^2_{1,j}x^1_{j,2}$ $=[0.4,-0.5,2.3]\cdot$ $[1.0,0.942,0.891]=$ 1.978	$\xi^2_{1,3}=\sum_{j=1}^3 w^2_{1,j}x^1_{j,3}$ $=[0.4,-0.5,2.3]\cdot$ $[1.0,0.269,0.668]=$ 1.802

Table 10: Layer $L=2$: interconnections with previous layer $\xi^l_{i,p}=\sum_{j=1}^{N_{l-1}} w^l_{i,j}x^{l-1}_{j,p}$, $N_1=3$

Layer $L=2$	Data point $p=1$	Data point $p=2$	Data point $p=3$
Target	$x^d_{1,1}=-1.0$	$x^d_{1,1}=1.0$	$x^d_{1,1}=1.0$
MSE	$E_1=\frac{1}{2}(x^2_{1,1}-x^d_{1,1})^2$ $=\frac{1}{2}(-1.0-0.986)^2$ $=1.972$	$E_2=\frac{1}{2}(x^2_{1,2}-x^d_{1,2})^2$ $=\frac{1}{2}(1.0-1.978)^2=$ 0.478	$E_3=\frac{1}{2}(x^2_{1,3}-x^d_{1,3})^2$ $=\frac{1}{2}(1.0-1.802)^2=$ 0.322

Table 11: Layer $L=2$: MSE loss $E_p=\frac{1}{2}\sum_{j=1}^{N_L}(x^L_{j,p}-x^d_{j,p})^2$, $N_L=N_2=1$

$$\min_{w^l_{i,j}} E=\min_{w^l_{i,j}}\frac{1}{P}\sum_{p=1}^{P}E_p \tag{21}$$

Layer $L=2$	Data point $p=1$	Data point $p=2$	Data point $p=3$
Component $i=1$	$\delta^2_{1,1}=(x^2_{1,1}-x^d_{1,1})$ $\sigma'(\xi^2_{1,1})=(0.986+$ $1)\sigma'(0.986)=0.393$	$\delta^2_{1,2}=(x^2_{1,2}-x^d_{1,2})$ $\sigma'(\xi^2_{1,2})=(1.978-$ $1)\sigma'(1.978)=0.104$	$\delta^2_{1,3}=(x^2_{1,3}-x^d_{1,3})$ $\sigma'(\xi^2_{1,3})=(1.802-$ $1)\sigma'(1.802)=0.098$

Table 12: Layer $L=2$: deltas $\delta^L_{i,p}=(x^L_{i,p}-x^d_{i,p})\sigma'(\xi^L_{i,p})$

$l=1$	Data point $p=1$	Data point $p=2$	Data point $p=3$
$i=2$	$\delta^1_{2,1}=\left(\sum_{j=1}^1\delta^2_{j,1}w^2_{j,2}\right)$ $\sigma'(\xi^1_{2,1})=0.393\times$ $(-0.5)\times\sigma'(-0.6)=$ -0.045	$\delta^1_{2,2}=\left(\sum_{j=1}^1\delta^2_{j,2}w^2_{j,2}\right)$ $\sigma'(\xi^1_{2,2})=0.104\times$ $(-0.5)\times\sigma'(2.8)=$ -0.003	$\delta^1_{2,3}=\left(\sum_{j=1}^1\delta^2_{j,3}w^2_{j,2}\right)$ $\sigma'(\xi^1_{2,3})=0.098\times$ $(-0.5)\times\sigma'(-1.0)=$ -0.010
$i=3$	$\delta^1_{3,1}=\left(\sum_{j=1}^1\delta^2_{j,1}w^2_{j,3}\right)$ $\sigma'(\xi^1_{3,1})=0.393\times2.3\times$ $\sigma'(-0.7)=0.200$	$\delta^1_{3,2}=\left(\sum_{j=1}^1\delta^2_{j,2}w^2_{j,3}\right)$ $\sigma'(\xi^1_{3,2})=0.104\times2.3\times$ $\sigma'(2.1)=0.023$	$\delta^1_{3,3}=\left(\sum_{j=1}^1\delta^2_{j,3}w^2_{j,3}\right)$ $\sigma'(\xi^1_{3,3})=0.098\times2.3\times$ $\sigma'(0.7)=0.050$

Table 13: Layer $l=1$: deltas $\delta^l_{i,p}=\left(\sum_{j=1}^{N_{l+1}}\delta^{l+1}_{j,p}w^{l+1}_{j,i}\right)\sigma'(\xi^l_{i,p})$, $N_2=1$

Learning rate $\eta = 1$.

$L = 2$	Source $j = 1$	Source $j = 2$	Source $j = 3$
$i = 1$	$\Delta w^2_{1,1} = -\frac{1}{3}\sum_{p=1}^{3} \delta^2_{1,p} x^1_{1,p} = -\frac{1}{3}[0.393, 0.104, 0.098] \cdot [1.0, 1.0, 1.0] = -0.198$	$\Delta w^2_{1,2} = -\frac{1}{3}\sum_{p=1}^{3} \delta^2_{1,p} x^1_{2,p} = -\frac{1}{3}[0.393, 0.104, 0.098] \cdot [0.354, 0.942, 0.269] = -0.088$	$\Delta w^2_{1,3} = -\frac{1}{3}\sum_{p=1}^{3} \delta^2_{1,p} x^1_{3,p} = -\frac{1}{3}[0.393, 0.104, 0.098] \cdot [0.332, 0.891, 0.668] = -0.096$

Table 14: Layer $L = 2$: Weight changes $\Delta w^L_{i,j} = -\eta \frac{\partial E}{\partial w^L_{i,j}} = -\eta \frac{1}{P} \sum_{p=1}^{P} \delta^L_{i,p} x^{L-1}_{j,p}, P = 3$

$l = 1$	Source $j = 1$	Source $j = 2$	Source $j = 3$
$i = 2$	$\Delta w^1_{2,1} = -\frac{1}{3}\sum_{p=1}^{3} \delta^1_{2,p} x^0_{1,p} = -\frac{1}{3}[-0.045, -0.003, -0.010] \cdot [1.0, 1.0, 1.0] = 0.019$	$\Delta w^1_{2,2} = -\frac{1}{3}\sum_{p=1}^{3} \delta^1_{2,p} x^0_{2,p} = -\frac{1}{3}[-0.045, -0.003, -0.010] \cdot [1.0, 0, -1.0] = 0.012$	$\Delta w^1_{2,3} = -\frac{1}{3}\sum_{p=1}^{3} \delta^1_{2,p} x^0_{3,p} = -\frac{1}{3}[-0.045, -0.003, -0.010] \cdot [1.0, -2, 1.0] = 0.016$
$i = 3$	$\Delta w^1_{3,1} = -\frac{1}{3}\sum_{p=1}^{3} \delta^1_{3,p} x^0_{1,p} = -\frac{1}{3}[0.200, 0.023, 0.049] \cdot [1.0, 1.0, 1.0] = -0.091$	$\Delta w^1_{3,2} = -\frac{1}{3}\sum_{p=1}^{3} \delta^1_{3,p} x^0_{2,p} = -\frac{1}{3}[0.200, 0.023, 0.049] \cdot [1.0, 0, -1.0] = -0.050$	$\Delta w^1_{3,3} = -\frac{1}{3}\sum_{p=1}^{3} \delta^1_{3,p} x^0_{3,p} = -\frac{1}{3}[0.200, 0.023, 0.049] \cdot [1.0, -2, 1.0] = -0.068$

Table 15: Layer $l = 1$: Weight changes $\Delta w^l_{i,j} = -\eta \frac{\partial E}{\partial w^l_{i,j}} = -\eta \frac{1}{P} \sum_{p=1}^{P} \delta^l_{i,p} x^{l-1}_{j,p}, P = 3$

$L = 2$	Source $j = 1$ (bias)	Source $j = 2$	Source $j = 3$
$i = 1$	$w^2_{1,1} = w^2_{1,1} + \Delta w^2_{1,1} = 0.4 - 0.198 = 0.202$	$w^2_{1,2} = w^2_{1,2} + \Delta w^2_{1,2} = -0.5 - 0.088 = -0.588$	$w^2_{1,3} = w^2_{1,3} + \Delta w^2_{1,3} = 2.3 - 0.096 = 2.204$

Table 16: Layer $L = 2$: updated weights $w^l_{i,j} = w^l_{i,j} + \Delta w^l_{i,j}$

Layer $l = 1$	Source $j = 1$ (bias)	Source $j = 2$	Source $j = 3$
$i = 2$	$w^1_{2,1} = w^1_{2,1} + \Delta w^1_{2,1} = 0.4 + 0.019 = 0.419$	$w^1_{2,2} = w^1_{2,2} + \Delta w^1_{2,2} = 0.2 + 0.012 = 0.212$	$w^1_{2,3} = w^1_{2,3} + \Delta w^1_{2,3} = -1.2 + 0.016 = -1.184$
$i = 3$	$w^1_{3,1} = w^1_{3,1} + \Delta w^1_{3,1} = 0.7 - 0.091 = 0.609$	$w^1_{3,2} = w^1_{3,2} + \Delta w^1_{3,2} = -0.7 - 0.050 = -0.75$	$w^1_{3,3} = w^1_{3,3} + \Delta w^1_{3,3} = -0.7 - 0.068 = -0.768$

Table 17: Layer $l = 1$: updated weights $w^l_{i,j} = w^l_{i,j} + \Delta w^l_{i,j}$

C Google Research Tuning Playbook

Some notes from the Google Research Tuning Playbook [8]:

Start with the most popular **optimiser** for the problem at hand / start something simple as a baseline, e.g. with SGD with fixed momentum, or Adam with fixed $\epsilon, \beta_1, \beta_2$. Then, if necessary, be prepared to tune all hyperparameters of the chosen optimiser, and to also have to tune model parameters.

Batch size is the key factor in determining training time and compute. A higher batch size will often reduce training time - which is beneficial because it allows for more time for hyperparameter tuning and general reduced latency of the dev cycle.

Batch size is not a hyperparameter for validation set performance. As long as all hyperparameters are well-tuned, especially the learning rate, regularisation and sufficient training steps, the batch size should not impact results.

- With a smaller batch size, there is more noise due to the sample variance which can have a regularising effect.
- Larger batch sizes can be more prone to overfitting and may require stronger regularisation.

Changing the batch size requires tuning the hyperparameters again!

Run training jobs at batch sizes *increasing power of 2* for a small number of steps until the job exceeds memory. If the batch size doubles, throughput should also double if the accelerator memory is not yet saturated. This is the best max batch size; there is no point in increasing batch size if it increases training time.

The incremental tuning strategy: Start with a simple configuration and incrementally make improvements, based on strong evidence to avoid adding unnecessary complexity.

Exploration vs. exploitation: Exploration, e.g. with *quasi-random search*, should reveal the most essential hyperparameters to tune and sensible ranges for them. Then use Bayesian optimisation tools to automatically find the best hyperparameter configuration.

If training is *compute-bound*, training time comes down to how long one is willing to wait. If it is *not compute-bound*, training time is determined by overfitting, training for long enough to achieve the best generalisation (and then performing *retrospective optimal checkout selection*).

D Review Questions

D.1 Multilayer feedforward networks and backpropagation

Explain similarities and differences between biological and artificial neural networks. A neuron has three main parts: a cell body, branching extensions called dendrites for receiving input, an axon that carries the neuron's output to the dendrites of other neurons. It is estimated that humans have over 10^{11} neurons and 10^{14} synapses. Switching time is slower than in transistors but connectivity is higher than in supercomputers.

The McCulloch-Pitts model of a neuron consists of: incoming signals x_i, interconnection weights w_i, a bias term (i.e. threshold value) b, an activation a, nonlinearity $f(\cdot)$, and output y, s.t. $a = \sum_i w_i x_i + b$, $y = f(a)$.

This model of artificial neural networks gave rise to multilayer perceptrons (MLPs). Such neural networks obtain their information during training, and store the information in the weights. It learns a nonlinear map from given patterns, and is robus against inaccuracies in data. For example, consider the XOR problem which cannot be separated by a straight line and requires an MLP (cf. VC-dimension).

This paradigm is thus closer to the brain works than how traditional digital computers work (which are based on purely on logic, software algorithms and sequential processing, i.e. which are more rigid) Still, the brain remains much more complex (i.e. in the number of neurons) and energetically efficient (albeit at lower processing speeds than artificial neural networks).

What are critical design considerations in neural networks? The number of neurons, objective (optimiser, may include regularisation and other terms) / cost (minimiser, typically error / average loss over training set) function, training algorithm, how to avoid local minima, when to stop training, how to guarantee generalisation, how to deal with noisy data.

Explain similarities and differences between multilayer perceptron (MLP) and radial basis function (RBF) networks. A feedforward network is called so because there are no feedback connections in which outputs of the model are fed back into itself (as in recurrent neural networks).

MLPs primarily use nonlinear activation functions while RBF networks use radial basis functions as activation functions.

Nonlinear activation functions commonly used in MLPs include the rectified linear unit (ReLU, $\max(0, x)$) and sigmoid functions ($\sigma(x) = 1/(1 + e^{-x})$).[21] For input and output layer neurons, a linear characteristic is usually chosen.

The term "radial basis function" refers to a class of functions whose output depends only on the distance from a center point. These functions are radially symmetric around the center, where there is a peak: $y = \sum_{i=1}^{n_h} w_i h(\|x - c_i\|)$ where n_h is the number of hidden neurons, $c_i \in \mathbb{R}^m$ the centre points.

[21] Derivatives of $\sigma(\cdot)$: sigmoid $\sigma(x) = \frac{1}{1+\exp(-x)}$, $\sigma' = \sigma(1-\sigma)$, tanh $\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$, $\sigma' = 1-\sigma^2$, ReLU $\max(0, x)$, $\sigma' = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$

The most common RBF is the Gaussian activation function $y = \sum_{i=1}^{n_h} w_i e^{-\frac{\|x-c_i\|_2^2}{\sigma_i^2}}$. Others are inverse multiquadratics (IMQs) $\phi(r) = \frac{1}{\sqrt{1+(\varepsilon r)^2}}$ where r is the Euclidean distance between the input and a centre point, and ε controls the smoothness and width of the function. The inverse multiquadratic function decreases as the distance r increases but never reaches zero.

Park and Sandberg (1991) showed that it is possible to approximate any continuous nonlinear function by means of an RBF network. Neural networks with unbounded activation functions (e.g. ReLU) is a universal approximator [Sonoda & Murata, 2017]

What are advantages or disadvantages of multilayer perceptrons versus polynomial expansions? Both MLPs and polynomial expansions can be used to approximate functions.

But polynomial expansion is more strongly affected by the curse of dimensionality - where the number of possible configurations of the parameters is much larger than the number of training examples. The approximation error is $\mathcal{O}(\frac{1}{n_p^{2/n}})$; n_p = the number of terms in the expansion, n = the dimension of the input space.

Neural networks avoid the curse of dimensionality in the sense that *the approximation error is independent of the dimension of the input space* n (under certain conditions). The approximation error for an MLP with one hidden layer is $\mathcal{O}(\frac{1}{n_h}); n_h$ = the number of hidden units.

$$y = f(x_1, x_2) \,:\, y = a_1x_1 + a_2x_2 + a_{11}x_1^2 + a_{22}x_2^2 + a_{12}x_1x_2 + \ldots \text{ vs. } y = w^T \tanh(V \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \beta)$$

A neural network causes most interesting loss functions to become nonconvex. This means that neural networks are usually trained by using iterative, gradient-based optimisers that merely drive the cost function to a very low value, rather closed-form linear regression solvers or convex optimisation algorithms with global convergence guarantees used to train logistic regression or SVMs. Stochastic gradient descent applied to nonconvex loss functions has no such convergence guarantee and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialise all weights to small random values. The biases may be initialised to zero or to small positive values.

Explain the backpropagation algorithm. Backpropagation "is an efficient way to compute the gradient of an error function w.r.t. the parameters of the model, by working with intermediate variables and applying the chain rule". It is "a special case of a general technique in numerical analysis called automatic differentiation". [2]

Weights are initialised at random. The inputs are forward propagated through the network. The loss is calculated by comparing the ouptuts to the targets. The error contribution of each neuron is calculated by backpropagation through the layers, and the neurons' weights are adjusted proportionally.

What is a momentum term? Adding a momentum term, $\Delta w_{ij}^l(k+1) = -\eta\delta_{i,p}^l x_{j,p}^{l-1} + \alpha\Delta w_{ij}^l(k)$, smoothes the update trajectory.

What is the difference between online, offline learning with backpropagation?
In online learning the weights are updated after each new pattern.
In offline learning the weights are updated after a full (mini-)batch.

What is overfitting and early stopping? Overfitting is training set memorisation. However, what separates machine learning from optimisation is that we want the generalisation error, also called the test error, and defined as the *expected value of the error on a new input*, to be low as well.

The *i.i.d. assumption* enables us to describe the data-generating process with a probability distribution over a single example - which enables us to mathematically study the relationship between training error and test error. For some fixed $\boldsymbol{w}$, the expected training error is exactly the same as the expected test error, because both expectations are formed using the same dataset sampling process. Underfitting occurs when the model is not able to obtain a sufficiently low error on the training set. *Overfitting* occurs when the gap between the training error and test error, the *generalisation gap*, is too large.

The training set is used for training. The validation set is used for stopping - when the minimal error on the validation set is reached.

Explain Newton learning. Newton's method uses the inverse of the Hessian and the gradient to determine the optimal step. It converges quadratically, which is much faster than steepest descent.

$$\text{Taylor expansion: } f(x) = f(x_0) + J^T \Delta x + \frac{1}{2}\Delta x^T H \Delta x + \ldots, \quad \Delta x = x - x_0 \text{ (step)}$$

$$\text{Optimal step: } \frac{\partial f}{\partial(\Delta x)} = 0 \quad \Rightarrow \quad J + H\Delta x = 0 \quad \Rightarrow \quad \Delta x = -H^{-1}J$$

However, computation and inversion of the Hessian, which can be large and dense in high-dimensional problems, is more expensive computationally. Furthermore, the Hessian often has zero eigenvalues, i.e. it is singular (and therefore cannot be inverted).

Levenberg-Marquardt, quasi-Newton methods can be used to overcome these issues.

Explain Levenberg-Marquardt learning. The Levenberg-Marquardt algorithm combines the advantages of the Gauss-Newton and steepest decent methods. A *damping term* balances the *trade-off between fast convergence and stability.*

Imposing a constraint $\|\Delta x\|_2 = 1$ gives the Lagrangian

$$\mathcal{L}(x, \lambda) = f(x_0) + J^T \Delta x + \frac{1}{2}\Delta x^T H \Delta x + \frac{1}{2}\lambda(\Delta x^T \Delta x - 1), \quad \Delta x = x - x_0 \text{ (step)}$$

$$\text{Optimal step: } \frac{\partial \mathcal{L}}{\partial(\Delta x)} = 0 \quad \Rightarrow \quad J + H\Delta x + \lambda \Delta x = 0 \quad \Rightarrow \quad \Delta x = -[H + \lambda I]^{-1}J$$

The Hessian (second-order derivatives, scalar-valued functions, describing curvature) are *approximated* with the Jacobian (first-order derivatives, vector-valued functions, describing the rate of change in all directions)]of the error function. When the approximation is not sufficient, steepest descent is used as a fallback.

$$\begin{cases} \lambda = 0 & \text{Newton's method} \\ \lambda \gg & \text{Steepest descent} \end{cases}$$

The step size is based on the local curvature of the error function, allowing for faster convergence in regions of small curvature, and more stable updates in regions of large curvature.

Explain quasi-Newton learning. The Hessian matrix is approximated rather than computed. An example of quasi-Newton methods is the BFGS algorithm.

When the neural network contains many interconnection weights, it becomes hard to store the matrices (Hessian and Jacobian) into computer memory. For large scale neural networks, therefore, conjugate gradient methods are preferred.

Explain conjugate gradient learning. Conjugate gradient methods were developed for large-sale problems, for which it was becoming difficult to store the Hessian / Jacobian matrices in memory.

Conjugate gradient methods reduce the number of iterations required to converge by maintaining a set of *conjugate search directions* that are *orthogonal to each other* and have *no information overlap*.

In each iteration, the method selects a new direction that is a linear combination of the conjugate search directions and performs a line search to find the minimum along that direction. The step size is chosen such that the new direction remains conjugate to the previous directions.

What is mini-batch learning? Full gradients can be expensive to compute, and may lead to getting stuck in worse local minima. Therefore one often estimates the gradient on a smaller subset of the training data (a mini-batch).

$$w_{t+1} = w_t - \eta \sum_{i \in \mathcal{B}_t} \frac{\partial l_i}{\partial w}\Big|_{w=w_t}, \text{ where } \mathcal{B}(t) \text{ is a random training data subset at iteration } t$$

AdaGrad (adaptive gradient descent) is a stochastic gradient version with a learning rate for each parameter.

What is the Adam training algorithm? SGD with an additional momentum term m_t, and $\beta \in (0, 1)$:

$$w_{t+1} = w_t - \eta m_{t+1}, \quad m_{t+1} = \beta m_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial l_i}{\partial w}\Big|_{w=w_t}$$

Nesterov accelerated momentum (instead of at the current point, $w = w_t$) computes the gradients at the predicted point, $w = w_t - \eta m_t$:

$$w_{t+1} = w_t - \eta m_{t+1}, \quad m_{t+1} = \beta m_t + (1 - \beta) \sum_{i \in \mathcal{B}_t} \frac{\partial l_i}{\partial w}\Big|_{w=w_t - \eta m_t}$$

Adam (Adaptive moment estimation) considers the momentum and an elementwise squaring of the gradient of the loss. In this way good progress during the training process is obtained in every direction in the parameter space. It is usually used with mini-batches.

D.2 Classification and Bayesian Decision Theory

What are the limitations of a perceptron? A perceptron consists of just one neuron with a sign activation function: $y = \text{sign}(v^T x + b) = \text{sign}(w^T z)$.

A perceptron can only learn linear decision boundaries. Thus it can shatter an AND or OR gate, for example. To be able to shatter an exclusive-OR (XOR) gate, a nonlinear decision boundary needs to be learned, i.e. a MLP with one hidden layer is needed.

What does Cover's theorem tell us about linear separability? Consider N input variables in d dimensions. Then, if $N < d+1$, any labelling of points is linearly separable.

For two classes C_1, C_2, each possible assignment of all points in the dataset is called a dichotomy. For N points, there are 2^N possible dichotomies. For larger d it becomes likely that more dichotomies are linearly separable.

That is, the fraction of dichotomies that is linearly separable, for the two cases, is given by:

$$F(N,d) = \begin{cases} 1 & N < d+1 \\ \frac{1}{2^{N-1}} \sum_{i=0}^{d} \binom{N-1}{i} & N \geq d+1 \end{cases}$$

Let $N = 4, d = 2$. Then there are $2^4 = 16$ dichotomies. 14 of them are linearly separable - all except XOR.

Consider a microarray dataset with $d = 10^4, N = 100$ (i.e. large d s.t. $N < d+1$ and definitely linearly separable), and a fraud detection dataset with $d = 10, N = 10^6$ (i.e. large N s.t. $N \geq d+1$).

Nonlinear separation with MLPs:

- Perceptron: linear separation.
- One hidden layer: realisation of convex regions.
- Two hidden layers: realisation of non-convex regions.

Explain simple linear scaling vs. whitening for preprocessing inputs.

What is the receiver operating (ROC) curve? Goal: max AUC (area under curve). Sensitivity = TP / (TP + FN). Specificity = TN / (FP + TN). FP-Rate = 1 - Specificity = FP / (FP + TN).

How is Bayes' theorem used for classification and decision making? Consider a character recognition problem with classes C_1, C_2 corresponding to letters 'a' and 'b'.

Assuming that 'a' typically occurs three times as often as 'b', the *prior* probabilities $P(C_1) = 0.75, P(C_2) = 0.25$. The best classifier would hence be one that always predicts 'a', since $P(C_1) > P(C_2)$ and always predicting 'a' therefore minimises the probability of misclassification.

Suppose a feature x_1, which takes discrete values X_l, is measured. The *posterior* probability is given by the class conditional probability times the prior over the normalisation: $P(C_k \mid X_l) = \frac{P(X_l|C_k)P(C_k)}{P(X_l|C_1)P(C_1)+P(X_l|C_2)P(C_2)+\ldots} (= \frac{P(C_k,X_l)}{P(X_l)})$.

Using Bayes' theorem, the probability of misclassification can be minimised based on the posterior $P(C_k \mid X_l)$, e.g. assign point to class C_1 if $P(C_1 \mid x_1) > P(C_2 \mid x_1)$.

Prior $P(C_k) \rightarrow$ observed data $\frac{P(X_l|C_k)}{P(X_l)} \rightarrow$ posterior $P(C_k \mid X_l)$.

For continuous random variables in the feature space,

$$P(C_k \mid x) = \frac{p(x \mid C_k)P(C_k)}{p(x)} = \frac{p(x \mid C_k)P(C_k)}{\sum_{k=1}^{c} p(x \mid C_k)P(C_k)}, \quad k = 1, \ldots, c,\ x \in \mathbb{R}^d, \quad \sum_{k=1}^{c} P(C_k \mid x) = 1$$

$p(x)$ is the unconditional density. Class-conditional densities $p(x \mid C_k)$ are obtained, for example, by probability density estimation methods. If $p(x \mid C_k)$ has a parametrised form, then it is called a *likelihood* function; that is, the posterior is then the likelihood times the prior over the normalisation.

Selecting the class with maximum posterior probability, $k^* = \arg\max_{k=1,\ldots,c} P(C_k \mid x)$, minimises the probability of misclassification. Imagine two bell curves, then:

$$P(\text{err.}) = \int_{\mathcal{R}_2} p(x \mid C_1)P(C_1)\,dx + \int_{\mathcal{R}_1} p(x \mid C_2)P(C_2)\,dx \qquad P(\text{corr.}) = \sum_{k=1}^{c} \int_{\mathcal{R}_k} p(x \mid C_k)P(C_k)\,dx$$

Minimizing the probability of misclassification may not be the best criterion in some circumstances, e.g. it might be more serious when a tumor is classified as normal than a normal image as tumor.

Risk minimisation: Define a loss matrix L where L_{kj} is the penalty associated with assigning a pattern to class C_j when it belongs to C_k. The expected loss or risk then is $R_k = \sum_{j=1}^{c} L_{kj} \int_{R_j} p(x \mid C_k)\,dx$, so that the overall expected loss or risk is $R = \sum_{k=1}^{c} R_k P(C_k)$. Region R_j is chosen if $\sum_{k=1}^{c} L_{kj} p(x \mid C_k)P(C_k) < \sum_{k=1}^{c} L_{ki} p(x \mid C_k)P(C_k) \quad \forall i \neq j$.

What is a discriminant function? Surfaces of constant probability, i.e. constant $\Delta^2 = (x-\mu)^T\Sigma^{-1}(x-\mu)$, are hyperellipsoids with principal axes u_i given by $\sum u_i = \lambda_i u_i$, with u_i, λ_i eigenvectors and eigenvalues of Σ. If Σ is diagonal, then the components x are statistically independent because in that case $p(x) = \prod_{i=1}^{d} p(x_i)$.

Bishop book. Watch lecture again.

How can we use a mixture model network for density estimation? Density estimation is concerned with modelling a probability density $p(x)$ given data points $x_n \in \mathbb{R}^d$, $n = 1, \ldots, N$; difficult in higher dimensions ($d > 10$).

Estimate class-conditional densities $p(x \mid C_k)$ and apply Bayes' Theorem to find the posterior.

There are parametric, non-parametric and semi-parametric approaches. Techniques include maximum likelihood and Bayesian inference.

The main appeal of the maximum likelihood estimator is that it can be shown to be the best estimator asymptotically, as the number of examples $m \rightarrow \infty$, the maximum likelihood estimate converges to the true value of the parameter (consistency). Relative to maximum likelihood estimation, Bayesian estimation offers two important differences: first, unlike the maximum likelihood approach that makes predictions using a point estimate of θ, the Bayesian approach is to make predictions using a full distribution over θ; second, the prior in Bayesian inference has an influence by shifting probability mass density towards regions of the parameter space that are preferred a priori (in practice, often a preference for simpler or smoother models). Bayesian methods typically generalise much better when limited training data is available but typically suffer from high

computational cost when the number of training examples is large. Most operations involving the Bayesian posterior for most interesting models are intractable, and a point estiamte (e.g. MAP, PME) offers a tractable approximation which still gains some of the benefit of the Bayesian approach by allowing the prior to influence the choice of the point estimate. Note that additional information from the prior in the MAP estimate helps to reduce variance at the price of increased bias (in comparison to the ML estimate).

In maximum likelihood estimation, represent the density $p(x)$ for a dataset $\chi = \{x_1, \ldots, x_N\}$ by means of a parameter vector $\theta = [\theta_1; \ldots; \theta_M] \in \mathbb{R}^M$, which gives a parametrised density $p(x \mid \theta)$. Assuming the data are drawn independently from $p(x \mid \theta)$, then $p(\chi \mid \theta) = \prod_{n=1}^{N} p(x_n \mid \theta) = \mathcal{L}(\theta)$ gives the likelihood $\mathcal{L}(\theta)$ of θ for the data χ. The maximum likelihood, i.e. the θ which is most likely to give rise to the observed data, is obtained by means of the minimum negative log likelihood:

$$\min_{\theta} E,\ E = -\log \mathcal{L}(\theta) = -\log \big[\prod_{n=1}^{N} p(x_n \mid \theta)\big] = -\sum_{n=1}^{N} \log p(x_n \mid \theta)$$

Example of a multivariate normal distribution for $p(x \mid \theta)$ - biased if N is finite, unbiased as $N \to \infty$.

For mixture models, a reparametrisation by $\gamma_j \in (-\infty, \infty)$ is used to guarantee that $\sum_{j=1}^{M} P(j) = 1$, $0 \geq P(j) \geq 1$, which gives the softmax function: $P(j) = \frac{e^{\gamma_j}}{\sum_{j=1}^{M} e^{\gamma_j}}$

In Bayesian inference, i.e. $p(\theta \mid \chi) = \frac{p(\chi|\theta)p(\theta)}{p(\chi)}$, for likelihood $p(\chi \mid \theta) = \prod_{n=1}^{N} p(x_n \mid \theta)$, the normalisation factor $p(\chi)$ ensures that $\int p(\theta \mid \chi)\, d\theta = 1$.

What is Expectation-Maximisation of mixture models? The EM algorithm reformulates the cost function in terms of hidden variables.

D.3 Generalisation and Bayesian Learning

What is a generalisation error and the bias-variance trade-off? The bias of an estimator is defined as $\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}[\hat{\boldsymbol{\theta}}_m] - \boldsymbol{\theta}$; an estimator is (asymptotically) unbiased if $\lim_{m\to\infty} \mathbb{E}[\hat{\boldsymbol{\theta}}_m] = \boldsymbol{\theta}$. *Consistency* ensures that the bias induced by the estimator diminishes as the number of data examples grows, e.g. convergence in probability $\text{plim}_{m\to\infty}\hat{\boldsymbol{\theta}}_m = \theta$. (However, asymptotic unbiasedness does not imply consistency.)

Bias and variance measure two different sources of error in an estimator: bias measures the expected deviation from the true value of the function or parameter; variance provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause. Evaluating the MSE incorporates both the bias and the variance: $\text{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] = \text{Bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m)$. Estimators with small MSE manage to keep both their bias and variance somewhat in check.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting. When generalisation error is measured by the MSE (where bias and variance are meaningful components of generalisation error), *increasing capacity tends to increase variance and decrease bias.*

The goal of learning is not to memorize data but rather to model the underlying generator of the data, characterized by the joint probability density of inputs x and targets t, $p(x,t) = p(t \mid x)p(x)$.

Consider a data set of infinite size, i.e. $N \to \infty$, with cost

$$E = \lim_{N\to\infty} \frac{1}{2N} \sum_{n=1}^{N} \big(y(x_n; w) - t_n\big)^2 = \frac{1}{2} \iint \big(y(x_n; w) - t_n\big)^2 \, p(t, x)\, dt\, dx$$

At the minimum $w*$ of the error function, the output approximates the conditional average of the target data: $y(x_n; w) = \int t\, p(t \mid x)\, dt$, but intrinsic noise puts a harder limit on the achievable error:

$$E = \frac{1}{2} \int \Big(y(x; w) - \int t\, p(t \mid x)\, dt\Big)^2 p(x)\, dx + \frac{1}{2} \int \Big(\int t^2\, p(t \mid x)\, dt - \big(\int t\, p(t \mid x)\, dt\big)^2\Big)^2 p(x)\, dx$$

In practice, there is only one finite data set D. Consider the expectation over an ensemble of datasets

$$\mathbb{E}_D[\big(y(x) - \int t\, p(t \mid x)\, dt\big)^2] = \big(\mathbb{E}_D[y(x)] - \int t\, p(t \mid x)\, dt\big)^2 + \mathbb{E}_D\Big[\big(y(x) - \mathbb{E}_D[y(x)]^2\big)^2\Big]$$

High bias means the model makes strong assumptions and fails to capture important patterns in the data. High bias typically leads to underfitting.

High variance means the model is too flexible and learns noise or random fluctuations in the training data. High variance typically leads to overfitting.

For example, generate 100 data sets by sampling the true underlying function $f(x)$[22] and adding noise. Estimate the mappings $y_i(x)$ for $i = 1, 2, \ldots, 100$. The average response is $\bar{y}(x) = \frac{1}{100} \sum_{i=1}^{100} y_i(x)$. The square of the bias is $\sum_n \big(\bar{y}(x_n) - f(x_n)\big)^2$, the variance is $\sum_n \frac{1}{100} \sum_{i=1}^{100} \big(y_i(x_n) - \bar{y}(x_n)\big)^2$.

In the bias-variance trade-off one tries to minimise the sum of bias^2 + variance!

[22]The true underlying function would not be known in the real world.

Typically, in a small model structure (with a small effective number of parameters) there is low variance and high bias; while in a large model structure (with a large effective number of parameters) there is high variance and small bias.

How can one prevent overfitting? Regularisation is any modification we make to a learning algorithm that is intended to reduce its generalisation error but not its training error.

Regularisation, for example with a small value of $\lambda = 10^{-6}$ for the L2 regularisation term in Ridge regression, avoids oscillation in a high-order polynomial. (Setting λ too high, on the other hand, may result in too much regularisation.)

Early stopping when the minimal error on the validation set is reached.

Data augmentation is easiest for classification. It has been a particularly effective technique for a specific classification problem: object recognition. Injecting noise in the input to a neural network [22] can also be seen as a form of data augmentation.

What is the role of a regularisation term (or weight decay term)? The regularisation term is a penalty term added to the loss function during training, which penalises large parameter values or encourages sparsity in the model. This encourages the model to prioritise smaller weights (L2) or select only a subset of features (L1), thus reducing overfitting and improving generalisation.

Sparsity is the property of having a large number of zero or near-zero values in the parameters or features of the model. This allows for more efficient and more interpretable models because only a subset of features are actively contributing to the model's output.

Consider a regularised problem $\tilde{E} = E + v\Omega(w)$ with E the original cost function (e.g. MSE), v a positive regularisation constant, $\Omega(w) = \frac{1}{2}\sum_i w_i^2$ the regularisation term with unknown weights w. Suppose $E = 0$. Then $\frac{dw}{d\tau} = -\eta\frac{\partial \tilde{E}}{\partial w} = -\eta v w$ yielding exponentially *decaying weights* $w(\tau) = w(0)e^{-\eta v \tau}$, where τ is the iteration number.

Weight Elimination: this algorithm is more likely to eliminate weights (i.e. setting to zero) than the weight decay method. A drawback is the choice of the additional tuning parameter c.

What is the effective number of parameters? Thanks to regularization one can implicitly work with less parameters than the number of unknown interconnection weights. The effective number of parameters is $\sum_j \frac{\lambda_j}{\lambda_j + v}$.

The number of parameters refers to the total count of weights in a model while the effective number of parameters refers to the numbers of parameters that aren't close to zero so they have an actually impact on the output. The amount parameters and effective number of parameters might not match because regularisation techniques and vanishing gradients.

The effective number of parameters refers to the number of non-zero parameters that actively contribute to the model's predictions.

In a sparse model, the effective number of parameters is smaller than the total number of parameters in the model because many of the parameters are forced to be zero by regularisation.

What is the difference between least squares and ridge regression? How is this related to the bias-variance trade-off? *Least squares* regression aims to find

the line (in the case of linear regression, or a hyperplane in the case of multiple linear regression), that minimises the sum of the squared differences between the observed and predicted values.

$$\min_{\boldsymbol{x}} J_{\text{LS}}(\boldsymbol{x}) = \frac{1}{2}(A\boldsymbol{x} - B)^T(A\boldsymbol{x} - B), \quad A \in \mathbb{R}^{m\times n}, m > n, B \in \mathbb{R}^m, \boldsymbol{x} \in \mathbb{R}^n$$

$$\text{Condition for optimality } \frac{\partial J_{\text{LS}}}{\partial \boldsymbol{x}} = \mathbf{0} \quad \Rightarrow \quad A^TA\boldsymbol{x} - A^TB = \mathbf{0}$$

$$\Rightarrow \quad \boldsymbol{x}_{\text{LS}} = (A^TA)^{-1}A^TB = A^\dagger B, \quad A^\dagger = \text{ the pseudoinverse}$$

Ridge regression adds a regularisation term $\|\boldsymbol{x}\|_2^2 = \boldsymbol{x}^T\boldsymbol{x}$ to the least squares objective function, known as the L2 regularisation term. This penalty shrinks the coefficients towards zero[23], and thus reduces variance and overfitting.

$$\min_{\boldsymbol{x}} J_{\text{ridge}}(\boldsymbol{x}) = \frac{1}{2}e^Te + \frac{1}{2}\lambda\boldsymbol{x}^T\boldsymbol{x}, \quad e = (A\boldsymbol{x} - B) \text{ s.t. } e^Te \text{ is the squared error, } \lambda > 0$$

$$\text{Condition for optimality } \frac{\partial J_{\text{ridge}}}{\partial \boldsymbol{x}} = \mathbf{0} \quad \Rightarrow \quad A^TA\boldsymbol{x} + \lambda\boldsymbol{x} - A^TB = \mathbf{0}$$

$$\Rightarrow \quad \boldsymbol{x}_{\text{ridge}} = (A^TA + \lambda I)^{-1}A^TB$$

Note this is helpful when A^TA is ill-conditioned.[24] When A^TA is ill-inconditioned, its smallest eigenvalues are close to zero, making the matrix nearly singular. This causes the OLS solution to be highly sensitive to small changes in the data, resulting in large variances and unstable estimates. The addition of λI increases all eigenvalues by λ, ensuring none are too close to zero and thus improving the condition number of the matrix.

The L1 norm $\tilde{E} = E + v\sum_i |w_i|$ gives sparsity but non-differentiability of $|w_i|$, and is used, for example, in Lasso regression and compressed sensing methods.

The *bias-variance trade-off* is the trade-off between bias and variance as a function of the regularisation term. A large penalty decreases variance but leads to a larger bias, a small penalty decreases bias but increases variance.

Bias is the inability to capture the true relationship in the data. Variance is the difference between the performance on the training and test sets. Compare with overfitting!

[23] Unlike lasso regression, which can force some coefficients to be exactly zero (thus performing variable selection), ridge regression only reduces their magnitude, so all coefficients remain nonzero unless λ is infinite.

[24] An ill-conditioned matrix is a matrix for which small changes or errors in the input data (such as the coefficients or the right-hand side of a system of equations) can cause large changes in the solution. This means that the matrix is highly sensitive to numerical errors, making computations involving it unreliable or unstable. For example, when solving $A\boldsymbol{x} = \boldsymbol{b}$, if A is ill-conditioned, even a tiny change in $\boldsymbol{b}$ or in the entries of A can lead to a large difference in the solution $\boldsymbol{x}$. The degree of ill-conditioning is measured by the condition number of the matrix. The condition number is determined by the smallest and largest singular values of the matrix. The more extreme the ratio between the largest and smallest eigenvalues, the more ill-conditioned the matrix is. Ill-conditioning often arises when the matrix has columns (or rows) that are nearly linearly dependent or have vastly different scales

Explain cross-validation. As opposed to always using the same validation set, in (k-fold) cross-validation the training set is split into k folds, and each fold is used as the validation set for using the other $k-1$ folds to train.

Can prevent overfitting, and metrics will be more conservative (check learning theory).

- In K-fold cross-validation, the original sample is randomly partitioned into K subsamples. Of the K subsamples, a single subsample is retained as the validation data for testing the model, and the remaining K-1 subsamples are used as training data. The cross-validation process is then repeated K times (the folds), with each of the K subsamples used exactly once as the validation data. The K results from the folds then can be averaged (or otherwise combined) to produce a single estimation
- The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once
- Cross-validation gives more conservative validation metrics!?
- leave-one-out cross-validation is the same as a K-fold cross- validation with K being equal to the number of observations in the original sample!

Explain complexity criteria. Prediction error (PE) is training error plus complexity.

For linear models, for example the Akaike information criterion, $PE = MSE + \frac{W}{N}\sigma^2$ where N is the number of training data, W is the number of adjustable parameters, σ^2 is the variance of the noise on the data.

For nonlinear models, the generalised prediction error $GPE = MSE + \frac{\gamma}{N}\sigma^2$ where $\gamma = \sum_i \frac{\lambda_i}{\lambda_i + v}$ is the effective number of parameters, with λ_i the eigenvalues of the Hessian of the unregularised error, v the regularisation coefficient. (Eigenvalues $\lambda_i \ll v$ do not contribute to the sum.)

What are pruning algorithms? In order to improve the generalisation performance of the trained models one can remove interconnection weights that are irrelevant.
Optimal Brain Damage: considers error change due to small changes in weights. Has been used to prune networks with 10,000 interconnections by a factor of 4. Starting from a relatively large network architecture,

1. Train until a stopping criterion is satisfied, e.g. the error function is minimised.
2. Compute the saliency values (measuring relative importance of weights) for all interconnection weights.
3. Sort the weights by saliency and delete low-saliency weights.
4. Repeat until some stopping criterion is reached.

Optimal Brain Surgeon:

1. Train a relatively large network to a minimum of the error function
2. Calculate the inverse of the Hessian to determine which weights can be removed without introducing a lot of error

3. Remove the weights that give the smallest increase in error error and update the weight matrix (it changes dimension).

4. Adjust remaining weights to account for removal of the weights.

5. Go to 2 and repeat until some stopping criterion is reached

OBS has better performance than OBD because OBD completely removes the connection with low-saliency from the network while removing the influence of the weight. This helps to maintain structural integrity.

What is a committee network (ensemble learning)? In the committee network method, also known as ensemble learning or model averaging, multiple individual models, called committee members, are trained independently on the same task or dataset, using possibly different algorithms or parameter settings, and their predictions are combined to obtain the committee network's prediction.

Training many different networks and selecting only the best one (based on a validation set) wastes a lot of training effort. Moreover, generalisation on the validation set has a random component due to noise on the data. The performance of a committee network can be better than the performance of the best single network.

The average sum-of-squares error for a single network is $E_i = \mathbb{E}[(y_i(x) - f(x))^2] = \mathbb{E}[\varepsilon_i^2]$. The average error over multiple networks is $E_{\text{AVG}} = \frac{1}{L}\sum_{i=1}^{L} E_i = \frac{1}{L}\sum_{i=1}^{L}\mathbb{E}[\varepsilon_i^2]$ For a committee network,

$$y_{\text{COM}}(x) = \frac{1}{L}\sum_{i=1}^{L} y_i(x), E_{\text{COM}} = \mathbb{E}[\big(\frac{1}{L}\sum_{i=1}^{L}(y_i(x) - f(x))^2\big)^2] = \mathbb{E}[\big(\frac{1}{L}\sum_{i=1}^{L}\varepsilon_i\big)^2]$$

Variance is reduced by averaging over many networks:

$$\text{By Cauchy's inequality, } \big(\frac{1}{L}\sum_{i=1}^{L}\varepsilon_i\big)^2 \leq \sum_{i=1}^{L}\varepsilon_i^2 \quad \Rightarrow \quad E_{\text{COM}} \leq E_{\text{AVG}}$$

Ensemble learning can be further improved by using a *weighted average* committee network.

What is a double descent curve? The classical U-shaped risk curve arises from the bias-variance trade-off. The double-descent risk curve incorporates the U-shaped risk curve (i.e. the classical regime) with the observed behaviour from using high-complexity function classes (i.e. the "modern" interpolating regime), separated by the interpolation threshold (i.e. the peak in the middle).

The area to the left of the minimum of the classical risk curve represents underfitting, the area to the right overfitting.

The area to the left of the double descent peak represents this classical, under-parametrised regime. The training risk reaches zero at the interpolation threshold. To the right of the interpolation threshold there is the modern over-parametrised interpolating regime. Note that under the modern regime the training risk is already and remains at zero.

What is the role of the prior distribution in Bayesian learning of neural networks? The prior represents knowledge or assumptions about the model parameters *prior* to observing the data. It provides a way of specifying beliefs about the likely values of the parameters based on prior experience or domain knowledge.

It can be chosen based on prior beliefs, or can be non-informative and expressing minimal prior knowledge. The choice of prior distribution reflects the prior assumptions about the parameters' values and their uncertainty.

Consider a model H given by $y(x; w)$, i.e. parametrised by parameter vector w, e.g. the interconnection weights of a MLP. The posterior $p(w \mid D)$ is the likelihood times the prior $p(w)$ over the evidence:

$$P(w \mid D, H) = \frac{P(D \mid w, H)P(w \mid H)}{P(D \mid H)}$$

Taking the log, the log posterior equals the log likelihood + the log prior - the log evidence.

The objective is $min_w M(w) = \beta E_D(w) + \alpha E_W(w)$ with $E_D(w) = \frac{1}{2}\sum_{n=1}^{N}\left(t^{(n)} - y(w^{(n)}; w)\right)^2$ and $E_W(w) = \frac{1}{2}\sum_j w_j^2$, i.e. keep also the weights small when minimising training error! Equivalently, $max_w -M(w) = -\beta E_D(w) - \alpha E_W(w)$.

Consider $-M(w)$ as the log posterior, $-\beta E_D(w)$ as the log likelihood, $-\alpha E_W(w)$ as the log prior. Then, using normalisation factors, one can consider $\exp(-M(w))/Z_M$ as the posterior, $\exp(-\beta E_D(w))/Z_D$ as the likelihood, $\exp(-\alpha E_W(w))/Z_W$ as the prior.

What is Occam's razor principle? Occam's razor states that one should not only try to minimise error but also keep the model complexity as low as possible: "Simple models should be preferred". Bayes Theorem embodies Occam's razor automatically!

Occam's razors says that the simplest explanation should be the one most preferred. For us, this would mean that we prefer the simplest model. Simple models tend to make precise predictions. Complex models by their nature , are capable of making a greater variety of predictions.

Consider two alternative models H_1, H_2 and data D. Then

$$\frac{P(H_1 \mid D)}{P(H_2 \mid D)} = \frac{P(D \mid H_1)}{P(D \mid H_2)}\frac{P(H_1)}{P(H_2)}$$

Suppose $P(H_1) = P(H_2)$ but H_1 is a simpler model that makes a more limited range of predictions $P(D \mid H_1)$ over a shorter range C_1. The more complex model H_2 is able to predict a larger variety of data sets, $C_2 > C_1$. If the data fall in region C_1, the simpler model H_1 is more probable because in interval C_1 the evidence $P(D \mid H_1) > P(D \mid H_2)$ so that $\frac{P(H_1|D)}{P(H_2|D)} = \frac{P(D|H_1)}{P(D|H_2)} > 1$.

What is the difference between parameters and hyper-parameters when training multilayer perceptrons? In the context of Bayesian learning (posterior, likelihood, prior, evidence), there are three types of unknowns: parameters w (the unknown interconnection weights), hyperparameters α, β (regularisation constants to be determined), the number of hidden units (leading to different models H_i). That is, there are three levels of inference:

1. params w: $\max_w \text{Post.} = \frac{\text{Likelih.}\times\text{Prior}}{\text{Evidence}} \equiv P(w \mid D, \alpha, \beta, H) = \frac{P(D|w,\alpha,\beta,H)P(w|\alpha,\beta,H)}{P(D|\alpha,\beta,H)}$

2. hyperparams α, β: $\max_{\alpha,\beta}$ Post. $= \frac{\text{Likelih.}\times\text{Prior}}{\text{Evidence}} \equiv P(\alpha, \beta \mid D, H) = \frac{P(D|\alpha,\beta,H)P(\alpha,\beta|H)}{P(D|H)}$

3. model comparison: $\max_{H_1,H_2,\ldots}$ Post. $= \frac{\text{Likelih.}\times\text{Prior}}{\text{Evidence}} \equiv P(H \mid D) = \frac{P(D|H)P(H)}{P(D)}$

The prior in level 1 is $P(w \mid \alpha, H) = \exp(-\alpha E_W(w))/Z_W \propto \exp(-\alpha \sum_j w_j^2)$. The larger the value for α, the more emphasis on making the weights small.

Parameters are the variables of model that are learned from the training data. The values that the model adjusts during the training to minimize the loss function. These include weights and biases associated with the connections between neurons.

Hyperparameters are the configurations of the neural network that are set before the training beings. These are not learned but set by the developer. Hyperparameter include the learning rate, number of hidden layers, number of neurons in each layer, etc.

How does one characterise uncertainties on predictions in a Bayesian learning framework? Instead of providing a single point estimate, Bayesian methods offer a distribution of predictions, called error bars, allowing for a more comprehensive understanding of the uncertainty associated with the model's output.

Bayesian prediction involves marginalisation over the uncertainty at all levels:

$$P(t^{(N+1)} \mid D) = \sum_{H_i} \int d\alpha\, d\beta \int d^k w P(t^{(N+1)} \mid w, \alpha, \beta, H) P(w, \alpha, \beta, H \mid D)$$

Assuming fixed values α, β and a local linearisation of the output, one has a predictive distribution with mean $y(x^{(N+1)}; w_{\text{MP}})$ and variance $\sigma^2_{t|\alpha,\beta} = J^T A^{-1} J + \frac{1}{\beta}$.

What is the purpose of automatic relevance determination? What is the role of the hyper-parameters?

- Input selection: Which are the most relevant inputs in order to explain the data with respect to the considered model?
- Different models can lead to different conclusions about relevance of inputs: Note that the obtained relevance is not valid in an absolute sense, but only relative with respect to the considered model.
- A choice of C for automatic relevance determination:

$$C(x, z) = \theta_1 \exp\left(-\frac{1}{2} \sum_{i=1}^{n} \frac{(x_i - z_i)^2}{\sigma_i^2} \right) + \theta_2$$

  where $x, z \in \mathbb{R}^n$ and x_i, z_i denote the i-th component of these vectors, θ_1, θ_2 are constants.

Interpretation: σ_i large: irrelevant to the model; σ_i small: i-th input is relevant for the considered model.

With support vector machines, an RBF kernel can be used to determine the relevance of inputs:

$$K(x, z) = \exp\big(-(x - z)^T S^{-1} (x - z) \big)$$

where $S = \text{diag}([s_1^2; \ldots; s_n^2])$. Small values of $1/s_i^2$ indicate that the corresponding input is not very relevant in the chosen model.

D.4 Time Series Prediction, Nonlinear Modelling and Control

How can a multilayer perceptron be used for time series prediction? A multilayer perceptron can be used if we prepare the data correctly. We organise the time series data into input-output pairs where the input is a sequence of past observations and the output is the predicted value for the next time step. We use these input-output pairs to train the MLP.

$$\hat{y}_{k+1} = f(y_k, y_{k-1}, \ldots, y_{k-p}) = w^T \tanh(V y_{k|k-p} + \beta), \text{ where } y_{k|k-p} = \left[y_k, y_{k-1}, \ldots, y_{k-p}\right]^T$$

Training as a feedforward network (e.g. by backpropagation) with training error to be minimised: $\min_{w,V,\beta} \frac{1}{2N} \sum_{k=p+1}^{p+N} (y_{k+1} - \hat{y}_{k+1})^2$ where $y_k = \hat{y}_k + e_k$.

Model selection considerations (for which a validation set or k-fold CV can be used):

- the number of lags p
- the number of hidden nodes in the hidden layer V
- the choice of regularisation constant (in case weight decay term used in objective)

Although you can use an MLP for time series prediction, it can be challenging. They are limited by their inability to retain memory of past inputs. This can lead to difficulties in accurately predicting future values. If the multilayer perceptron is recurrent neural network, we also have the problem of the vanishing gradient.

How can one use neural networks in different model structures for system identification? We can use neural networks in system identifications like time series by training on the input and output data and letting the neural net find out how the output is correlated to the input data. Doing this should allow one to discover the working of the system and predict outputs.

1. System characterisation: definition of inputs, outputs, states.
    - White box models: model equations obtained by applying physical laws, parameters have a physical meaning.
    - Black box models: predictive model able to establish the relation between inputs and outputs of the system, estimated parameters do not have any physical meaning.
    - Grey box models: aspects of both.
2. Choice of a model structure:
    - Linear time-invariant I/O model structures: ARX, ARMAX, OE, Box-Jenkins.
    - Non-linear I/O model structures: NARX, NARMAX, NOE.
    - Deterministic state space models: linear, nonlinear.
    - Stochastic state space models: linear, nonlinear.
3. Parametrisation of the model: represent model in unknown parameters.
    - Parametrisations by neural nets: I/O (MLP, RBF), state space (MLP) models.

Explain the use of dynamic backpropagation. Dynamic backpropagation is a modification of the standard backpropagation algorithm used in feedforward networks, which takes into account the time dependency of the data in recurrent networks.

Computation of the gradient of a cost function defined on a recurrent network is more complicated. It involves a sensitivity model, which is another dynamical system derived from the model.

Another approach is Werbos' backpropagation through time.

What is an LSTM neural network? LSTMs consist of units with input, output and forget gates. W, U are the interconnection matrices, b the bias terms, d is the number of inputs, h the number of hidden units, $\circ$ is the elementwise product.

$$\begin{cases} f_t &= \text{sigmoid}(W_f x_t + U_f h_{t-1} + b_f) \\ i_t &= \text{sigmoid}(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \text{sigmoid}(W_o x_t + U_o h_{t-1} + b_o) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \tanh(W_c x_t + U_c h_{t-1} + b_c) \\ h_t &= o_t \circ \tanh(c_t) \end{cases}$$

LSTMs address the vanishing gradient problem.

Explain the use of neural networks for control applications. A control system is a mechanism that manages, commands, directs, or regulates the behavior of other devices or systems using control loops. It operates by comparing the current state or output of a system (the process variable, PV) to a desired state (the setpoint, SP). The system then applies a control action-often through feedback-to minimize the difference (error) between the actual and desired values, thereby achieving the intended behavior or performance.

There are two main types of control systems:

- Open-loop control: The control action is independent of the output (no feedback).
- Closed-loop control (feedback control): The control action depends on the output, using feedback to reduce the error and stabilize the system

Neural networks can be used in control applications with 3 different approaches:

- Behaviour cloning: a form of imitation learning where an agent learns to perform a task by observing and mimicking the actions of an expert. This is typically achieved by collecting a dataset of input-output pairs, where the input is the state or observation (such as images from a car's camera) and the output is the expert's action (such as steering angle or acceleration). The agent then uses supervised learning to train a model that maps inputs to outputs, effectively "cloning" the expert's behaviour (e.g. learning to drive a car)
- Model-based control of a system: first estimate a (neural network) model and then design a neural controller based on the estimated model (e.g. backpropagation through the network).

- Model-free control: For Example, reinforcement learning methods - In this approach, the neural network is used to learn a control policy through interactions with the environment. The neural network acts as the policy network or value function approximator in reinforcement learning algorithms. The system is controlled based on feedback and rewards obtained from the environment, allowing the neural network to learn optimal control strategies. An example could be training a neural network to control a robot arm to perform a specific task without explicitly modeling the dynamics of the system.

Consider a system defined by $x_0, x_{k+1} = f(x_k, u_k)$ with control constraints $u_k \in U$ and objective $\max J = \psi(x_N) + \sum_{k=0}^{N-1} l(x_k, u_k)$ for fixed N and with l a positive real-valued function.

Dynamic programming derives the optimal solution from the optimal return function $V(x,k) : V(x_N, N) = \psi(x_N), V(x,k) = \max_{u \in U} \big(l(x,u) + V\big(f(x,u), k+1\big)\big)$

Q-Learning is an incremental approximation to dynamic programming. The objective is to find a control rule that maximises at each step the expected discounted sum of future reward in a finite-state, finite-action MDP. Estimate a real-valued function $Q(x,a)$ of state x, action a, which is the expected discounted sum of future reward for performing an action a in state x and performing optimally thereafter:

$$Q(x,a) = \mathbb{E}[r_k + \gamma \max_b Q(x_{k+1}, b) \mid x_k = x, a_k = a]$$

In Deep Q-Learning (DQN; Mnih et al., 2015 Atari paper), $Q(x,a;\theta)$ is parametrised by a vector θ of a neural network to learn the function.

D.5 Associative Memories

Feedforward neural networks are static systems. Recurrent neural networks are dynamical systems with feedback connections.

A dynamical system is a mathematical model in which a function describes how the state of a system evolves over time within a given space. At any moment, the system has a state (often represented as a point in a state or phase space), and the evolution rule (usually a set of equations) determines how the state changes as time progresses. The evolution can be deterministic (future states are uniquely determined by the current state) or stochastic (randomness influences the evolution). Formally, a dynamical system is often defined as a tuple (T, X, Φ), where: T is the set representing time (e.g., real numbers for continuous time, integers for discrete time), X is the state space, Φ is the evolution function that describes how the state changes with time.

- continuous-time dynamical system: $\frac{dx(t)}{dt} = f(x(t))$ with initial condition $x(0) = c$ for state vector $x \in \mathbb{R}^n$.
- discrete-time dynamical system: $x_{k+1} = f(x_k)$, $x_0 = c$ where k is a discrete time index.

In nonlinear systems there may be multiple or a single unique equilibrium point. Equilibrium points may be locally stable or unstable.

- continuous-time equilibrium points $x*$ satisfy $\frac{dx}{dt} = 0$ or $f(x*) = 0$.
- discrete-time equilibrium points $x*$ satisfy $x* = f(x*)$.

What is the working principle of associative memories from a dynamical systems point of view? Associative memory stores recognised patterns as equilibrium points of the system.

When the network is trained, it creates a valley in a virtual energy landscape for every saved pattern. It does this by changing the weights between connections so that when it is in such a pattern, the energy function is at its lowest.

When the trained network is fed with a partial or noisy pattern, it can retrieve the stored pattern by self-organising to the closest learned pattern. Like when dropping a ball down a slope, the ball rolls until it reaches a place where it is surrounded by uphills. In the same way, the network makes its way towards lower energy and finds the closest saved pattern.

What is the Hebb rule for storing patterns in associative memories and why does it work? The Hebbian learning rule is based on the idea that "neurons that fire together, wire together". The rule states that the weight between two neurons should be increased if both neurons are active at the same time and decreased if one neuron is active while the other is not. The Hebbian rule provides a mechanism for modifying synaptic strengths based on the co-activation of neurons. It works by strengthening the connections between neurons that consistently fire together, facilitating associative memory storage and retrieval.

What determines the storage capacity in associative memories? The storage capacity is limited by the ability to retrieve stored information without errors and the potential interference between stored patterns. The ability to retrieve the information without errors is dependent on the amount of neurons:

$$p_{\max} = \frac{N}{4 \log N}$$

The more neurons, the more patterns can be stored without errors.

When solving the TSP problem using a Hopfield network, how are cities and a tour being represented? To solve TSP using a Hopfield network, we first need to define the energy function. In the case of TSP, the energy function is defined as the sum of the distances between adjacent cities along the route, where the route must visit every city exactly once. When this sum is lowest, so the energy function as well, we have found the shortest distances.

What are the main differences between modern Hopfield networks and classical Hopfiled networks? How does it update patterns? Does it use the synchronous update rule or the asynchronous one? Classical Hopfield networks use Hebbian learning through a sum of outer products of the N binary patterns, and the weight matrix then stores the patterns, which can be retrieved.

Modern Hopfield networks use an energy function (e.g. polynomial with storage capacity $C \cong \alpha_a d^{a-1}$, exponential with storage capacity $C \cong 2^{d/2}$)

Compared to the traditional Hopfield Networks, the increased storage capacity now allows pulling apart close patterns. We are now able to distinguish (strongly) correlated patterns, and can retrieve one specific pattern out of many. [3, 14]

How is the discrete modern Hopfield network generalized to the continuous one? How does it update patterns? Does it use the synchronous update rule or the asynchronous one? It can be shown that the energy function generalises to the attention formula. Such a Hopfield layer for deep learning can be used for pattern retrieval tasks. [21]

Explain the connection between the continuous modern Hopfield network and the attention mechanism.

D.6 Vector Quantisation, Self-Organising Maps, Regularisation

What is the aim of vector quantisation? The aim of vector quantisation is to compress or represent a set of continuous-valued vectors using a smaller set of discrete-valued vectors, known as prototype vectors. The goal is to approximate the original data while reducing the amount of information required to represent it.

It achieves this by partitioning the input vector space into regions and assigning a representative prototype vector to each region. The selected prototype vectors should capture the essential characteristics of the input vectors. Vector quantisation introduces some level of error in the reconstructed data due to the approximation process. The aim is to control this error and strike a balance between compression and reconstruction quality.

Competitive learning (or stochastic approximation) VQ algorithm: Consider data $x(k)$ for $k = 1, 2, \ldots$ and initial centres $c_j(0)$ for $j = 1, 2, \ldots, s$ centres (or prototypes).

1. Determine the nearest centre $c_j(k)$ to data point $x(k)$, e.g. squared error loss $J = \arg\min_l \|x(k) - c_l(k)\|$, called a **competition** between centres.
2. **Update** the centre: $c_j(k+1) = c_j(k) + \gamma(k)\big(x(k) - c_j(k)\big)$, $k := k + 1$ where $\gamma(k)$ meets the conditions for stochastic approximation.

Note: The process of building the codebook in vector quantization is essentially the same as running k-means: the centroids found by k-means become the codebook vectors in VQ. Assigning each data point to the nearest centroid (in k-means) is the same as quantizing a vector to its closest codebook entry (in VQ). In both cases, the objective is to minimize distortion (the average squared distance between data points and their representatives).

The algorithm minimises a distortion measure. The partition regions of a vector quantiser are non-overlapping and cover the entire input space $\mathbb{R}^n$. The optimal vector quantiser is the Voronoi partition.

How is vector quantisation related to self-organising maps? Self-organising maps, also known as Kohonen maps,

- try to represent the underlying density of the input by means of prototype vectors (similar to VQ).
- project the higher dimensional input data to a map of neurons (also called nodes or units) such that the data can be visualised, typically a projection to a 2-dimensional grid of neurons.

In this way the SOM compresses information while preserving the most important topological and metric relationships of the primary data items on the display.

Neurons in fact have two positions: the prototypes c_j in input space $\mathbb{R}^n$, and the map coordinates z_j in output space $\mathbb{R}^2$. For the grid, often a hexagonal grid is chosen, and also rectangular grids are an option.

SOM, like VQ, use competitive learning:

1. At time k, determine the winning unit (also called the best matching unit).
2. Update all units.
3. Decrease the learning rate $\beta(k)$ and width $\sigma(k)$.

How are RBF networks related to regularisation theory?

How can unsupervised learning be helpful for training RBF networks?

What are fuzzy models? The Gaussian RBF function can also be used in fuzzy models as a membership function.

In fuzzy modelling, one works with "vague statements" such as "It is cold" instead of a crisp statement like "The temperature is 7 degrees".

The membership function expresses the uncertainty that one has about facts. Classically, uncertainty is often expressed in terms of probability. Probabilities and membership grades are not the same. Fuzzy logic is the theory of how to compute with such vague statements, instead of with the usual binary 0/1 logic. For example: "If the heating power is high, then the temperature will increase fast".

For control applications it has the desirable property that models and control strategies can be understood and translated into words such that human operators can understand what it's doing and, moreover, that human knowledge can be incorporated within these systems.

D.7 Support Vector Machines and Kernel-Based Models

Assume that data for two classes were each sampled from a Gaussian distribution, and both distributions have equal covariance matrices, i.e. $C_1 = C_2 = C$. This implies that only their means are different.

LDA: A new point x belongs to the first class if $P(C_1 \mid x) > P(C_2 \mid x)$. The equation $P(C_1 \mid x) = P(C_2 \mid x)$ describes all points on the boundary between the two classes.

By Bayes' Theorem,

$$P(C_1 \mid x) = \frac{P(C_1)P(x \mid C_1)}{P(C_1)P(x \mid C_1) + P(C_2)P(x \mid C_2)}$$

Assuming equal probability of selecting each class, i.e. $P(C_1) = P(C_2)$, it follows from the equation $P(C_1 \mid x) = P(C_2 \mid x)$ that $P(x \mid C_1) = P(x \mid C_2)$. Thus,

$$\begin{aligned}
\frac{1}{2\pi\sqrt{\det C}} d^{-\frac{1}{2}(x-m_1)^T C^{-1}(x-m_1)} &= \frac{1}{2\pi\sqrt{\det C}} d^{-\frac{1}{2}(x-m_2)^T C^{-1}(x-m_2)} \\
-\frac{1}{2}(x-m_1)^T C^{-1}(x-m_1) &= -\frac{1}{2}(x-m_2)^T C^{-1}(x-m_2) \\
(m_2-m_1)^T C^{-1} x - \frac{1}{2}(m_2-m_1)^T C^{-1}(m_2+m_1) &= 0 \\
w^T x + b &= 0
\end{aligned}$$

Separating hyperplane with $w = (m_2 - m_1)^T C^{-1}$, $b = -\frac{1}{2}(m_2 - m_1)^T C^{-1}(m_2 + m_1)$.

The Linear Discriminant Function: $g(x) = \boldsymbol{w}^T \boldsymbol{x} + b$, defines a hyper-plane in the feature space.

The unit-length normal vector of the hyper-plane is: $\boldsymbol{n} = \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}$.

How to minimise the classification error rate? Large Margin Classifier: The linear discriminant function (classifier) with the maximum margin is the best.

Why is it best? Robust to outliers and, thus, strong generalisation ability.

Margin is defined as the width that the boundary could be increased with before hitting a data point. The support vectors are on the margin.

Given a set of data points $\{(\boldsymbol{x}_i, y_i)\}, i = 1, 2, \ldots, n$ where

- for $y_i = +1$, $\boldsymbol{w}^T \boldsymbol{x}_i + b > 0$
- for $y_i = -1$, $\boldsymbol{w}^T \boldsymbol{x}_i + b < 0$

With a scale transformation on both w and b, the above is equivalent to:

- for $y_i = +1$, $\boldsymbol{w}^T \boldsymbol{x}_i + b \geq 1$
- for $y_i = -1$, $\boldsymbol{w}^T \boldsymbol{x}_i + b \leq -1$

That is,

- The line through the positive support vector: $\boldsymbol{w}^T \boldsymbol{x}^+ + b = 1$
- The line through the middle: $\boldsymbol{w}^T \boldsymbol{x}^+ + b = 1$
- The line through the negative support vector: $\boldsymbol{w}^T \boldsymbol{x}^- + b = -1$

The margin width is:

$$\begin{aligned} M &= (\boldsymbol{x}^+ - \boldsymbol{x}^-) \cdot \boldsymbol{n} \\ &= (\boldsymbol{x}^+ - \boldsymbol{x}^-) \cdot \frac{\boldsymbol{w}}{\|\boldsymbol{w}\|} \\ &= \frac{2}{\|\boldsymbol{w}\|} \end{aligned}$$

What are advantages of support vector machines in comparison with classical multilayer perceptrons? SVM's are more easily interpreted than classical multiplayer perceptron's. they also maximize the distance between classes. SVM's are also more computational efficient, and less effected by outliers. SVMs also don't have the problem of ending up in local minima solutions like MLP's do. Some of the advantages:

- Effective in high-dimension space, by using support vectors they can avoid the curse of dimensionality.
- Kernel trick for Nonlinear data
- Guarantees global optimum solutions
- Computationally efficient and less affected by outliers.

What is the kernel trick in support vector machines? The kernel trick is used to transform data that isn't linearly separable to another dimension so we can separate it without actually calculating the features vectors in that dimension, thus making it more computational efficient.

What is a support vector? A support vector is a data point that captures the essential information needed to discriminate between different classes. They provide the information for constructing an optimal decision boundary and determining the boundaries for new data points. SVM's try to maximize the distance between 2 differently classified points to get accurate predictions. These vectors then get stored and used to make predictions about the test data. By using support vectors, which are a subset of the training data, we can computationally efficiently deal with high-dimension datasets.

What is a primal and a dual problem in support vector machines? SVM's can be defined in two ways, one is the primal form and the other one is the dual form. The dual form is an alternative formulation of the SVM optimisation. It involves transforming the primal problem into a constrained optimisation problem, expressed in terms of Langrange multipliers. The primal form optimises the objective function subject to a set of constraints while the the dual form create a Langrangian function that combines the objective function and the constraints. The primal form is preferred when we don't need to apply the kernel trick to the data, the dataset is large but the dimension of each data point is small. Dual form is preferred when data has a huge dimension and we need to apply the kernel trick.

D.8 Nonlinear PCA and (Stacked) Autoencoders

What is nonlinear PCA? What is an encoder and a decoder? Aim of PCA: decrease dimensionality by mapping vectors $x \in \mathbb{R}^n$ to $z \in \mathbb{R}^m$ with $m < n$.

When using Principal component analysis (PCA) we calculate the covariance matrix of all the data, then calculate the eigenvalues and eigenvectors. Afterwards we project the data on the most important eigenvectors (the ones with the highest eigenvalues as these hold the most information). By doing this projection we essentially project the data in a lower dimension which we can feed to for example neural networks to learn from in a less computational expensive way. In nonlinear PCA we use kernel functions for mapping to higher dimension, which is useful for data points who can not be described with linear function like circles.

Given data $\{x_i\}_{i=1}^N$, $x_i \in \mathbb{R}^n$ with assumed zero mean and ellipsoidal shape, potentially in a high-dimensional input space, find projected variables $w^T x_i$ with *maximal variance*:

$$\max_w \mathbb{E}[(w^T x)^2] = w^T \mathbb{E}[xx^T] w = w^T C w$$

where $C = \mathbb{E}[xx^T]$ is the covariance matrix. For N data points,

$$C \approx \frac{1}{N} \sum_{i=1}^{N} x_i x_i^T$$

Problem: the optimal solution for w is unbounded. A common choice for an additional constraint therefore is to impose $w^T w = 1$. The problem formulation then becomes:

$$\max_w w^T C w \text{ subject to } w^T w = 1$$

This is a constrained optimisation problem solved by taking the Lagrangian $\mathcal{L}(w; \lambda) = \frac{1}{2} w^T C w - \lambda(w^T w - 1)$ with Lagrange multiplier λ. Its solution is given by the eigenvalue decomposition $Cw = \lambda w$ with $C = C^T \geq 0$, obtained from setting $\partial \mathcal{L}/\partial w = 0$, $\partial \mathcal{L}/\partial \lambda = 0$.

- The eigenvalues λ_i are real and positive.
- The eigenvectors u_1, u_2 are orthogonal with respect to each other.
- The maximal variance solution is the direction u_1 corresponding to $\lambda_{\max} = \lambda_1$.
- Important: $\mu = 0$ (data should be centered beforehand).
- A point x is mapped to z in the lower-dimensional space by $z_j = u_j^T x$ where u_j are the eigenvectors corresponding to the m largest eigenvalues and $z = [z_1\, z_2 \ldots z_m]^T$.
- The error resulting from the dimensionality reduction is characterised by the neglected eigenvalues, i.e. $\sum_{i=m+1}^{n} \lambda_i$.

When is it a good choice? For example, when the two largest eigenvalues are much larger than the others, reducing the original space to a two-dimensional space makes sense. (Dimensionality reduction to one dimension would give a projection of the data onto vector u_1 which would remove all ability to discriminate between the two classes.)

The reconstruction problem: The point of PCA is projecting the data to a lower dimension. By doing this we don't use all of the eigenvectors which causes us to lose

some information, as the original matrix cannot be fully recovered. This is a trade off we take and in exchange we can use the used eigenvectors as features which are less computationally expensive.

Consider dimensionality reduction with $x \in \mathbb{R}^n, z \in \mathbb{R}^m, m \ll n$.

- $z = G(x)$ is the encoder mapping, $\tilde{x} = F(z)$ is the decoder mapping.
- The objective is the squared distortion error (reconstruction error):

$$\min E = \frac{1}{N}\sum_{i=1}^{N} \|x_i - \tilde{x}_i\|_2^2 = \frac{1}{N}\sum_{i=1}^{N} \|x_i - F(G(x_i))\|_2^2$$

- Information bottleneck: $x \to G(x) \to z \to F(z) \to \tilde{x}$
- (Linear PCA: linear mappings F, G; nolinearlinear PCA: nonlinear mappings F, G, e.g. parametrised by MLPs, and learn the interconnection matrices of MLP1 and MLP2 by minimising the reconstruction error.)

What is Oja's rule for a linear neuron in PCA analysis? Oja's learning rule is an unsupervised learning algorithm for dimensionality reduction or feature extraction based on the Hebbian learning principle.

It iteratively adapts the weights of the neural network to learn to represent the directions of maximum variance in the dataset, the principal components. It can be seen as a neural network-based approach to compute the principal components (while PCA calculates them using matrix operations).

Consider a linear neuron with output y and inputs ξ_j: $y = \sum_j w_j \xi_j = w^T \xi$. Assuming time dependency, i.e. $y(t) = w(t)^T \xi(t)$, Oja's learning rule states:

$$w(t+1) = w(t) + \eta y(t)\big(\xi(t) - y(t)w(t)\big)$$

Averaging over time t:

$$\begin{aligned}\mathbb{E}[w(t+1) - w(t)] &= \mathbb{E}[\eta y(t)\big(\xi(t) - y(t)w(t)\big)] \\ &= \mathbb{E}[\eta w(t)^T \xi(t)\big(\xi(t) - w(t)^T \xi(t) w(t)\big)], \qquad y(t) = w(t)^T \xi(t)\end{aligned}$$

Oja's rule is susceptible to getting trapped in local optima.

What is a reconstruction error for an autoencoder? Autoencoder: unsupervised (i.e. unlabelled training examples) learning algorithm with target values equal to inputs: $y^{(i)} = x^{(i)} \in \mathbb{R}^n, i = 1, 2, \ldots$.

Key point: limiting the numer of hidden units, i.e. imposing a *sparsity* constraint.

$$\min \|\text{input} - \text{decoder}(\text{code})\|^2 + \text{sparsity}(\text{code})$$

Aim: learning a *compressed representation* of the input (similar to a PCA reconstruction problem).

Explain the pre-training and fine-tuning steps when combining an autoencoder with a classifier. During the pre-training we train autoencoders to condense the data/learn important features, this is done unsupervised. When starting to fine-tune we append a classifier and add the training data assignments to use the features learned from the autoencoders to predict the classification of test data.

1. Pre-training: train a sparse autoencoder on the unlabelled data.
2. Fine-tuning: append a classifier which uses the features a learned by the autoencoder's hidden layer as inputs, and fine-tune the overall classifier by further supervised learning.

Give motivations for considering the use of more hidden layers in multilayer feedforward neural networks. Neural networks with one hidden layer are universal approximators but we can't say how many neurons may be needed in that hidden layer. For example: One hidden layer and n inputs may require 2^n hidden units which is a very large number. But if we add more hidden layers the amount of neurons needed may be more moderate. So not exponential but rather polynomial. These extra layers will allow the neural network to capture the underlying features and thus work out more complex tasks.

Deep network: multiple hidden layers to determine more complex features of the input. Important to use a nonlinear activation function (multiple layers of linear functions would compute only a linear function of the input).

- Compactly represent a significantly larger set of functions
- There are functions which a k-layer network can represent compactly (with a number of hidden units that is polynomial in the number of inputs), that a (k-1)-layer network cannot represent, unless it has an exponentially large number of hidden units.
- One can learn part-whole decompositions (feature hierarchies): e.g. layer 1: learn to group pixels in an image to detect edges, layer 2: group together edges to detect longer contours, (detect simple parts of objects), layer 3,4,...: group together contours or detect complex features
- Cortical computations in the brain also have multiple layers of processing: cortical area V1, cortical area V2

What are possible difficulties for training deep networks? Some of the most known ones are overfitting, vanishing/exploding gradients, and computational difficulties.

Further they also get less interpretable and explainable making them more and more of a black box model.

Supervised learning using labeled data by applying gradient descent (e.g. backpropagation) on deep networks usually does not work well. Too many bad local minima occur.

Gradients becoming very small in deep networks for randomly initialized weights. When using backpropagation to compute the derivatives, the gradients that are backpropagated rapidly decrease in magnitude as the depth of the network increases (called diffusion of gradient problem).

Greedy layer-wise training works better.

Explain stacked autoencoders. An autoencoder is a type of neural network that is trained (unsupervised) to reproduce its input data as its output. It consists of two parts: an encoder and a decoder. The encoder transforms the input data into a compressed representation, with typically a lower dimensionality than the input. So it can be more efficiently used. The decoder attempts to reconstruct the original data from the compressed representation.

Stacked autoencoders: greedy layer-wise training

Stacked autoencoders are several autoencoders stacked on top of each other. The first autoencoder learns the primary features of the data. These primary features are then given to the next autoencoder which in turn learns the secondary features. We can continue doing this to keep on condensing the input as far as we want. We can then use a classifier such as softmax to classify the output of last autoencoder. By condensing the data before feeding it to a classifier, we benefit from dimensionality reduction and thus increased efficiency.

1. train a sparse autoencoder (SAE) on the inputs $x^{(i)}$ to learn primary features $h^{(1)}$ on the input.

2. use the primary features as input to next SAE to learn secondary features $h^{(2)}$ on these primary features.

3. use secondary features as input to a classifier.

4. combine all layers together to form a SAE with 2 hidden and a classification layer, and apply fine-tuning.

D.9 Convolutional Neural Networks (CNNs)

Images have a very large number of input variables (precluding the use of fully connected networks) and behave similarly at every position (leading to the idea of parameter sharing).

For 28×28 images like in MNIST fully connected layers would be feasible. But for 96×96 images, i.e. with 10^4 inputs, if learning 100 features, then 10^6 weights would need to be learned.

Therefore there is a need for *locally connected networks*. Early visual system in the brain: neurons in the visual cortex have localized receptive fields (they respond only to stimuli in a certain location).

Explain feature extraction using the convolution and max-pooling operations. Natural images have the property of being stationary (i.e. statistics of one part of the image are the same as any other part). This suggests that the features that we learn at one part of the image, e.g. over a smaller 8×8 patch sampled randomly from a 96×96 image, can also be applied to other parts of the image, i.e. the learned 8×8 feature detector can be applied anywhere in the image.

Convolutions originate from linear time invariant (LTI) systems. A general MIMO (multi-input, multi-output) convolution form, with $c_{\text{in}}, c_{\text{out}}$ the number of input and output channels and $\boldsymbol{h}_{i,j}$ the convolution kernel that contributes to the i-th channel output by convolving with the j-th input channel image, is:

$$\boldsymbol{y}_i = \sum_{j=1}^{c_{\text{in}}} \boldsymbol{h}_{i,j} * \boldsymbol{x}_j, \quad i = 1, \ldots, c_{\text{out}}$$

Classically, features were user-defined, e.g. a fixed convolution kernel for edge detection. Now, deep feature learning is used.

1. Train a sparse autoencoder on small $a \times b$ patches x_{small}, learning k *features* $f = \sigma(W^{(1)} x_{\text{small}} + b^{(1)})$.
2. Take the learned 8×8 features and convolve them with the larger image, which gives a $k \times (r - a + 1) \times (c - b + 1)$ array of *convolved features*.
3. *Pool* (given a window, stride, padding) the convolved features to limit the number of features.

Example: Input image filtered by four 5×5 convolutional kernels which create 4 feature maps - for (softmax[25]) classification -, after the feature extraction.

Convolution layers apply filter or kernel to the input image to extract features, such as edges or patterns. When we apply this filter over the input image we calculate the dot product and then put the output of the dot product unto a feature map. Each filter produces a new feature map, which highlights a specific pattern in the input data. The size of the feature map is smaller than the input data, as the filter moves over the input, pixels are shared and produce smaller outputs.

Pooling layers down sample the feature maps by reducing their dimensions, while retaining the most important information. This helps to reduce the number of parameters and computation required in the layers that come next. This is done by taking small subregions of the feature map and summarizing their contents into a single value.

[25] A softmax classifier produces normalized class probabilities, and has a probabilistic interpretation.

What is a ResNet architecture? A residual neural network takes short-cuts by jumping over some layers - inspired by pyramidal cells in the cerebral cortex.

Skip (or residual) connections result in a smoother loss surface, which facilitates learning and makes the final network performance more robust to minor errors in the parameters, so it will likely generalise better.

DenseNet uses residual connections to concatenate the outputs of earlier layers to later ones. The three-channel input image is processed to form a 32-channel representation. The input image is concatenated to this to give a total of 35 channels. This combined representation is processed to create another 32-channel representation, and both earlier representations are concatenated to create a total of 67 channels, and so on.

What is the meaning of invariance and equivariance?

- Invariance of a function $\boldsymbol{f}(\boldsymbol{x})$ to a transformation $\boldsymbol{t}(\boldsymbol{x})$: $\boldsymbol{f}(\boldsymbol{t}(\boldsymbol{x})) = \boldsymbol{f}(\boldsymbol{x})$
- Equivariance of a function $\boldsymbol{f}(\boldsymbol{x})$ to a transformation $\boldsymbol{t}(\boldsymbol{x})$: $\boldsymbol{f}(\boldsymbol{t}(\boldsymbol{x})) = \boldsymbol{t}(\boldsymbol{f}(\boldsymbol{x}))$

CNN training With a softmax, the average loss for a classification task is the cross-entropy loss between the target and estimated class distribution (provided y is normalised, as is done by the softmax). For regression, such as denoising in image processing $\mathbb{E}\|y - f_\Theta(x)\|_p^p$, $p = 1$ or $p = 2$ is minimised.

Data augmentation can be applied with creating new training instances (e.g. by applying geometric transformations such as mirroring, flipping, rotation, on the original image so that it doesn't change the label information).

With dropout regularization a neuron is temporarily "dropped" or disabled with a certain probability. All the inputs and outputs to some neurons will be disabled at the current iteration. The dropped-out neurons are resampled with a certain probability at every training step, so a dropped-out neuron at one step can be active at the next one [Ye 2022].

Bagging involves training multiple models and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory. Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

D.10 Generative Models

Discriminative models learn the decision boundary between the classes. Generative models model the probability distribution of each class.

Explain Restricted Boltzmann Machines. In a Boltzmann Machine, the probability of any configuration is directly related to its energy level. Lower energy configurations are more likely to occur than higher energy ones. By minimizing the energy of correct configurations, the machine essentially "learns" a probability distribution that represents the underlying patterns in the data. Boltzmann Machines view learning as a statistical process where different neuron configurations have different probabilities, and learning is all about maximizing the likelihood of the observed data.

A Boltzmann Machine is a fully connected Markov Random Field[26] of visible and hidden stochastic binary units. The objective is to minimise the overall energy of the system. A Restricted Boltzmann Machine is restricted in that it is a bipartite graph - there are only intra-layer connections (between hidden and visible units) but no interconnections between hidden units nor between visible units. This bipartite structure makes explicit marginalisation possible - which is why RBMs are more computationally efficient.

A restricted Boltzmann machine is a generative neural network that has no connections within a layer. In general, RBMs consist of 2 layers, a hidden layer and an input layer.RBMs were created because the connection in Boltzmann machines grow exponentially and thus are hard to train. RBMs are used in unsupervised learning tasks such as dimensionality reduction and data generation using Gibbs sampling. For more complex data we can use Deep Boltzmann machines which are several RBMs stacked on top of each other.

Explain Deep Boltzmann Machines. Deep Boltzmann Machines are a type of unsupervised learning models that are made of several layers of restricted Boltzmann machines. The difference between this DBM and DBN (deep belief networks) is that the connections between the layers at the bottom are also undirected. In these machines the surface layers represent the simple low-level features and the deeper layers represent the abstract, high-level features. The layers are trained one at a time.

What is Wasserstein training of Restricted Boltzmann Machines? Wasserstein distance in RBMs aims to reduce the distance between generated data and input data as much as possible. By using the Wasserstein distance we get less noise in the generated data which In turns makes the data more accurate. It is easily visualized by using it on image generations for example the digits dataset we used in our reports. Here we saw that the generated data with Wasserstein had less random white pixels.

What is the principle of GANs? Generative adversarial networks can be described as 2 separate neural nets. A generator which tries to generate "fake" input data and a discriminator which tries to distinguish fake and real input data. The two networks are trained together in a game-like process where the generator tries to generate more realistic data to fool the discriminator, while the discriminator tries to correctly classify the real and generated data.

[26] A Markov Random Field is an undirected, symmetric PGM that satisfies the Markov Property and can be factorised over cliques.

What is an ELBO in VAE? Given observed data X and latent variables Z, Bayesian inference seeks the posterior:

$$p(Z|X) = \frac{p(X|Z)\,p(Z)}{p(X)} \tag{22}$$

where

$$p(X) = \int p(X|Z)\,p(Z)\,dZ \tag{23}$$

Since $p(X)$ is generally intractable, we approximate $p(Z|X)$ with its variational approximation $q(Z)$ by minimizing:

$$\mathrm{KL}(q(Z) \,\|\, p(Z|X)) = \mathbb{E}_q\left[\log \frac{q(Z)}{p(Z|X)}\right] \tag{24}$$

Expanding with Bayes' rule:

$$\begin{aligned}
\mathrm{KL}(q(Z) \,\|\, p(Z|X)) &= \mathbb{E}_q\left[\log \frac{q(Z)}{p(Z|X)}\right] && (25)\\
&= \mathbb{E}_q\left[\log q(Z) - \log p(Z|X)\right] && (26)\\
&= \mathbb{E}_q\left[\log q(Z) - \log \frac{p(X,Z)}{p(X)}\right] && (27)\\
&= \mathbb{E}_q\left[\log q(Z) - \log p(X,Z) + \log p(X)\right] && (28)\\
&= \mathbb{E}_q[\log q(Z)] - \mathbb{E}_q[\log p(X,Z)] + \log p(X) && (29)
\end{aligned}$$

Thus,

$$\log p(X) = \underbrace{\mathrm{KL}(q(Z) \,\|\, p(Z|X))}_{\geq 0} + \mathbb{E}_q[\log p(X,Z)] - \mathbb{E}_q[\log q(Z)] \tag{30}$$

Defining the **Evidence Lower Bound (ELBO)**, the variational lower-bound for a single data point:

$$\mathcal{L}(q) = \mathbb{E}_q[\log p(X,Z)] - \mathbb{E}_q[\log q(Z)] = \mathbb{E}_q\left[\log \frac{p(X,Z)}{q(Z)}\right] \tag{31}$$

we have

$$\log p(X) = \mathrm{KL}(q(Z) \,\|\, p(Z|X)) + \mathcal{L}(q) \tag{32}$$

Since $\mathrm{KL}(q(Z) \,\|\, p(Z|X)) \geq 0$, $\mathcal{L}(q) \leq \log p(X)$. Maximizing $\mathcal{L}(q)$ tightens the bound and improves the approximation. Here:

- $\log p(X)$ is the log-evidence (model evidence), which is **fixed** for given data X.
- $\mathrm{KL}(q(Z) \,\|\, p(Z|X)) \geq 0$ is the Kullback–Leibler divergence between $q(Z)$ and the true posterior.
- $\mathcal{L}(q)$ is the Evidence Lower Bound (ELBO), a function of q alone.

Maximizing ELBO $\iff$ Minimizing KL Divergence Because $\log p(X)$ is a constant (with respect to q), as we increase $\mathcal{L}(q)$, the KL divergence must decrease:

$$\mathcal{L}(q) \uparrow \quad \Longrightarrow \quad \mathrm{KL}(q(Z) \,\|\, p(Z|X)) \downarrow \tag{33}$$

The optimal value is reached when $q(Z) = p(Z|X)$, so

$$\mathrm{KL}(q(Z) \,\|\, p(Z|X)) = 0 \tag{34}$$

and

$$\mathcal{L}(q) = \log p(X) \tag{35}$$

However: In practice, $p(Z|X)$ is intractable. KL cannot be minimized directly. But the ELBO involves only quantities that can be evaluated or estimated, due to our choice of $q(Z)$. optimising the ELBO therefore brings $q(Z)$ as close as allowed (by its form) to the true posterior.

Why is This a Good Approximation?

- **ELBO as a Bound:** $\mathcal{L}(q) \leq \log p(X)$ for all $q(Z)$.
- **Tightening the Bound:** Maximizing $\mathcal{L}(q)$ tightens this lower bound by making $q(Z)$ approach $p(Z|X)$ in the sense of KL divergence.
- **Best Approximation in the Family:** Since we can only choose $q(Z)$ from some tractable family (e.g., a factorized Gaussian), maximizing the ELBO finds *the closest $q(Z)$ to $p(Z|X)$ in that family*, as measured by KL divergence.

Summary:

> Maximizing the ELBO is equivalent to minimizing the KL divergence between our approximation $q(Z)$ and the true posterior $p(Z|X)$. Because the KL divergence is non-negative and $\log p(X)$ is fixed, optimising the ELBO yields the best possible approximation to the true posterior available within our chosen $q(Z)$ family:
>
> $$\boxed{q^*(Z) = \arg\max_{q(Z)} \mathcal{L}(q(Z)) = \arg\min_{q(Z)} \mathrm{KL}\big(q(Z) \,||\, p(Z|X)\big)}$$

In this way, maximizing the ELBO gives us the "closest" and most faithful approximation to the posterior distribution that we can achieve given computational constraints.

D.11 Normalisation

Normalisation originated from the batch normalisation technique (accelerates the convergence of stochastic gradient methods by reducing covariate shift). Batch normalisation was further extended to various forms of normalisation, such as layer norm, instance norm, and group norm.

What is batch normalisation? Batch normalization [12] was originally proposed to reduce the internal covariate shift and improve the speed, performance, and stability of artificial neural networks. This problem is particularly severe for deep networks because small changes in shallower hidden layers are amplified as they propagate through the network, causing a significant shift in deeper hidden layer.

Batch normalisation accelerates the convergence of stochastic gradient methods by reducing covariate shift.

Reduce these undesirable shifts and normalise values in each layer by centering and rescaling to have 0 mean and 1 variance.

For each hidden unit h_i compute $h_i \leftarrow \frac{g}{\sigma}(h_i - \mu)$ where g is a variable, $\mu = \frac{1}{H}\sum_{i=1}^{H} h_i$ and $\sigma = \sqrt{\frac{1}{HW}\sum_{i=1}^{H}(h_i - \mu)^2}$ over the mini-batch.

This reduces covariate shift (i.e. gradient dependencies between each layer) and therefore fewer training iterations are needed, speeding up training.

Advantages:

- Can increase learning rate without gradients vanishing or exploding.
- Appears to have a regularisation effect, thus there is less of a need use other types of regularisation like dropout to reduce overfitting.
- More robust toward different initialisation schemes and learning rates.

What is layer normalisation? Layer normalisation computes the mean and standard deviation along the channel and image direction without considering the mini-batch.

Thus, each sample has a different normalisation operation, allowing arbitrary mini-batch sizes to be used.

Experimentally layer normalisation performs well for RNNs.

What is instance normalisation? Instance normalisation normalises the feature data for each sample and channel.

$$\boldsymbol{y}_c = \frac{\gamma_c}{\sigma_c}(\boldsymbol{x}_c - \mu_c \mathbf{1}) + \beta_c \mathbf{1}, \qquad c = 1, \ldots, C$$

where γ_c, β_c are trainable parameters for each channel c, and

$$\mu_c = \frac{1}{HW}\mathbf{1}^T \boldsymbol{x}_c, \quad \sigma_c^2 = \frac{1}{HW}\|\boldsymbol{x}_c - \mu_c \mathbf{1}\|$$

What is style transfer? Style transfer: convert content images into a stylised image that is guided by a certain style image.

Adaptive instance normalisation (AdaIN): calculate γ_c, β_c as the standard deviation and mean value of the style image:

$$\beta_c^{(s)} = \frac{1}{HW}\mathbf{1}^T \boldsymbol{s}_c, \quad \gamma_c^{(s)} = \big(\frac{1}{HW}\|\boldsymbol{s}_c - \beta_c^2\mathbf{1}\|\big)^{\frac{1}{2}}$$

AdaIN is a special case (when the covariance matrix is diagonal) of the Whitening and Colouring Transform (WCT): $B_{x,s}$ as in AdaIN, whitening transform T_x, colouring transform T_s.

$$Y = XT_xT_s + B_{x,s}$$

StyleGAN: generator generates content feature vectors from random noise. AdaIN layer combines content and style features to generate more realistic features. (Fundamentally different from GANs where the fake image is only generated by a content generator.)

D.12 Attention

Cognitive neuroscience: selectively focusing on one aspect of information.

- Ionotropic receptors can open or close a channel so that different types of ions can migrate in and out of the cell.
- Metabotropic receptors only indirectly influence the opening and closing of ion channels, but they have more of a prolonged effect.

What is an attention mechanism, query and key? Let x_n be the number of neurotransmitters that bind to the n-th synapse. G-proteins generate the secondary messengers that bind to the ion channel at the m-th synapse with sensitivity q_m. The number of G-proteins generated at the n-th synapse is proportional to the metabotropic receptor's sensitivity k_n. The total amount of ion influx from the m-th synapse then is

$$y_m = \sum_{n=1}^{N} q_m k_n x_n, \quad m = 1, \dots, N$$

$$\boldsymbol{y} = qk^T x = Tx, \quad T = qk^T, \quad \text{rank}(T) = 1$$

Attention mechanism: for a particular key k, changing the query q gives a different activation pattern y. Thus, by decoupling key and query, neuronal activation patterns can be dynamically adapted.

In neural networks, the row vector output at the m-th pixel is

$$(\boldsymbol{y}_m)^T = \sum_{n=1}^{N} a_{mn} \boldsymbol{x}^n, \quad m = 1, \dots, N \quad \boldsymbol{q}_m \in \mathbb{R}^d, \boldsymbol{k}_n \in \mathbb{R}^d, \boldsymbol{x}_n \in \mathbb{R}^C$$

$$a_{mn} = \frac{\exp\big(\text{sim-score}(\boldsymbol{q}_m, \boldsymbol{k}_n)\big)}{\sum_{n'=1}^{N} \exp\big(\text{sim-score}(\boldsymbol{q}_m, \boldsymbol{k}_{n'})\big)}$$

- Dot product: $\langle \boldsymbol{q}_m, \boldsymbol{k}_n \rangle$
- Scaled dot product: $\langle \boldsymbol{q}_m, \boldsymbol{k}_n \rangle / \sqrt{d}$
- Cosine similarity: $\frac{\langle \boldsymbol{q}_m, \boldsymbol{k}_n \rangle}{\|\boldsymbol{q}_m\| \|\boldsymbol{k}_n\|}$

In dot product attention, the query and key vectors are usually generated using linear embeddings. Often one is interested in a smaller-dimensional value vector $\boldsymbol{v}_n$ Attention computes a weighted average of a set of values, where the weights are derived by comparing the query vector to a set of keys with a scoring function: Query, compare with each Key via scoring function, run each through softmax to get attention weights that *sum to one*, weight values by their weights to get the output.

Attention is a neural architecture that mimics the retrieval of a value v_i for a query q based on a key k_i in a database: "if the key matches the query, return its associated value".

$$\text{Attention}(q, \boldsymbol{k}, \boldsymbol{v}) = \sum_i \text{Similarity}(q, k_i) \times v_i$$

The only difference between database retrieval and attention in a sense is that in database retrieval we only get one value as input, but here we get a weighted combination of values.

Similarity can be a simple dot product of the query and the key. It can be scaled dot product, where the dot product of q and k, is divided by the square root of the dimensionality of each key, d. These are the most commonly used two techniques to find the similarity.

Often a query is projected into a new space by using a weight matrix W, and then a dot product is made with the key k. Kernel methods can also be used as a similarity.

Masking to prevent later tokens from influencing earlier ones.

What is self-attention? The advantage of the attention mechanism is to separate control of query and key vectors.

In a self-attention block, both query and key are obtained from the same dataset. It tries to extract which part of the (single) input signal needs to be focused.

Self-attention can learn the relationship between a pixel and all other positions, to handle details better.

Self-attention GAN (SAGAN): self-attention layers so both generator and discriminator can better capture relationships between spatial regions. The self-attended feature vector $\boldsymbol{o}^n$ is generated at the n-th pixel location by the linear combination of the value vectors *across the whole image* by weighting the elements of the attention map A:

$$Y = AV = AXW_V, \qquad O = YW_O$$

The receptive field of the self-attention map is an overall image, which makes image generation more effective; but can be computationally expensive (matrix multiplication of $N \times N$ attention map A).

To address the computational complexity, channel attention techniques have been developed: the squeeze and excitation network (SENet) squeezes using average pooling, followed by the excitation step which involves a neural network.

1. Input $X_{D\times N}$ with N input vectors $\boldsymbol{x}_n$ as its column vectors. Operated on separately by query, key, value matrices.
2. $Q_{D\times N} = \beta_q \mathbf{1}^T + \Omega_q X$. (biases and weights)
3. $K_{D\times N} = \beta_k \mathbf{1}^T + \Omega_k X$.
4. Attention $A_{N\times N} = \text{softmax}(K^T Q)$.
5. $V_{D\times N} = \beta_v \mathbf{1}^T + \Omega_v X$.
6. Output same dimension as input $Y_{D\times N} = V \cdot \text{softmax}(K^T Q)$.

The overall self-attention computation is nonlinear because the attention weights are themselves nonlinear functions of the input. This is an example of a hypernetwork, where one network branch computes the weights of another.

What is multi-head attention? Multi-head ≈ "multiple *concatenated* feature maps". The concatenation is vertical. Another linear combination is used to recombine them:

$$\Omega_c\big([A_1(X), A_2(X)]^T\big)^T$$

Masked multi-head attention: multi-head where some values are masked (i.e. probabilities of masked values are nullified to prevent them from being selected). When decoding, an output value should depend only on previous outputs (not future outputs). Hence we mask future outputs.

$$\text{Attention}(Q, K, V) = \text{softmax}\big(\frac{Q^T K}{\sqrt{d_k}}\big)V$$

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\big(\frac{Q^T K + M}{\sqrt{d_k}}\big)V$$

Note: softmax produces a distribution that adds up to one, therefore important to add M inside softmax term.

Interestingly, it appears that most of the heads can be pruned after training without critically affecting the performance (Voita et al., 2019); it has been suggested that their role is to guard against bad initializations.

What is cross-domain attention? The flow of computation is the same as in standard self-attention, but the queries are now calculated from the decoder embeddings (while the keys and values are from the encoder embeddings).

1. Decoder input $X_{\text{dec},D\times N_{\text{dec}}}$ with N_{dec} input vectors $\boldsymbol{x}_n$ as its column vectors.
2. Encoder input $X_{\text{enc},D\times N_{\text{enc}}}$ with N_{enc} input vectors $\boldsymbol{x}_n$ as its column vectors.
3. $Q_{D\times N_{\text{dec}}} = \beta_q \mathbf{1}^T + \Omega_q X_{\text{dec}}$.
4. $K_{D\times N_{\text{enc}}} = \beta_k \mathbf{1}^T + \Omega_k X_{\text{enc}}$.
5. Attention $A_{N_{\text{enc}}\times N_{\text{dec}}} = \text{softmax}(K^T Q)$.
6. $V_{D\times N_{\text{enc}}} = \beta_v \mathbf{1}^T + \Omega_v X_{\text{enc}}$.
7. Output same dimension as decoder input $Y_{D\times N_{\text{dec}}} = V \cdot \text{softmax}(K^T Q)$.

Attentional GAN (AttnGAN) for text-to-image generation: Query vector generated from image areas, key vector generated from word features by sentence embedding. Selects the word-level condition to generate different parts of the image.

Machine translation: a sequence-to-sequence task that requires an encoder-decoder model with cross-attention. The encoder learns the language embedding (i.e. the structural role of each word within a sentence) from the tokenised input, decoder performs language translation (uses encoder embeddings to generate key vectors, which are combined with the query vector that is generated from the target language).

D.13 Transformers

What are transformers? In order to learn long-range dependencies between words within a sentence, the self-attention mechanism is used on the encoder.

In addition to self-attention, there are:

- residual connections,
- layer normalisation,
- a feed-forward NN,
- additional units of encoder blocks.

The sharing of parameters in the network architecture facilitates the massively parallel processing of the transformer, and also allows the network to learn long-range dependencies just as effectively as short-range dependencies. However, the lack of dependence on token order becomes a major limitation when we consider sequential data, such as the words in a natural language, because the representation learned by a transformer will be independent of the input token ordering.

In contrast to RNNs, processes the entire sequence in parallel, to capture longer-distance dependencies in sequences. But the model does not have any notion of position (which is important for grammar and semantics!) for each word: it is equivariant with respect to input permutations.

Positional encoding: should output a unique encoding of the position of each word in a sentence and easily generalise to longer sentences.

Let n be the desired position in an input setence, and d the encoding dimension (an even number!). The position encoding vector, which is added to the word embedding vector to obtain a position-encoded word embedding vector $\boldsymbol{x}_n \in \mathbb{R}^d$ to feed into the Transformer's self-attention model, is given by

$$\boldsymbol{p}_n = \begin{bmatrix} \sin(\omega_1 n) \\ \cos(\omega_1 n) \\ \sin(\omega_2 n) \\ \cos(\omega_2 n) \\ \vdots \\ \sin(\omega_{\frac{d}{2}} n) \\ \cos(\omega_{\frac{d}{2}} n) \end{bmatrix} \in \mathbb{R}^d, \quad \omega_k = \frac{1}{10,000^{2k/d}}, \qquad \boldsymbol{x}_n \leftarrow \boldsymbol{x}_n + \boldsymbol{p}_n$$

Transformers have a low computational complexity per layer, and much of the computation can be performed in parallel using the matrix form. Since every input embedding interacts with every other, it can describe long-range dependencies in text. Ultimately, the computation scales quadratically with the sequence length

What are the inputs in a transformer? One problem with using a fixed dictionary of words is that it cannot cope with words not in the dictionary or which are misspelled. It also does not take account of punctuation symbols or other character sequences such as computer code. An alternative approach that addresses these problems would be to work at the level of characters instead of using words, so that our dictionary comprises upper-case and lower-case letters, numbers, punctuation, and white-space symbols such

as spaces and tabs. A disadvantage of this approach, however, is that it discards the semantically important word structure of language, and the subsequent neural network would have to learn to reassemble words from elementary characters. It would also require a much larger number of sequential steps for a given body of text, thereby increasing the computational cost of processing the sequence. We can combine the benefits of character-level and word-level representations by using a pre-processing step that converts a string of words and punctuation symbols into a string of tokens, which are generally small groups of characters and might include common words in their entirety, along with fragments of longer words as well as individual characters that can be assembled into less common words (Schuster and Nakajima, 2012).

Tokenisation: As an example, a technique called byte pair encoding that is used for data compression, can be adapted to text tokenization by merging characters instead of bytes (Sennrich, Haddow, and Birch, 2015). The process starts with the individual characters and iteratively merges them into longer strings. The list of tokens is first initialized with the list of individual characters. Then a body of text is searched for the most frequently occurring adjacent pairs of tokens and these are replaced with a new token. To ensure that words are not merged, a new token is not formed from two tokens if the second token starts with a white space. The process is repeated iteratively

BERT vs. GPT Encoder models like BERT exploit transfer learning.

Bidirectional Encoder Representations from Transformers (BERT): can be used for different purposes and languages by simply changing the training scheme.

1. Pre-training to learn parameters using self-supervised learning: guess the masked word in an input sentence (where 15% of words are masked with a specific token `[MASK]`) from the embedded output in the same place.

2. Fine-tuning using supervised learning tasks.

Text classification: In BERT, a special token known as the classification or `<cls>` token is placed at the start of each string during pre-training. For text classification tasks like sentiment analysis (in which the passage is labeled as having a positive or negative emotional tone), the vector associated with the `<cls>` token is mapped to a single number and passed through a logistic sigmoid which contributes to a standard binary cross-entropy loss.

Decoder models like GPT are autoregressive generators.

Generative Pre-Trained Transformer (GPT): The goal is similar to BERT pre-training, generating the next word - hence the name. Scaling greatly improves task-agnostic, few-shot performance. This is a major reason for the sucess of GPT (e.g. GPT-3 with 175B parameters vs. BERT with 340M parameters): its massive architecture makes generative pre-training even more powerful than fine-tuning.

The process of choosing tokens from these probability distributions is known as **decoding**. It is not computationally feasible to try every combination of tokens in the output sequence ($\mathcal{O}(K^N)$ which grows exponentially with the length of the sequence; greedy search has cost $\mathcal{O}(KN)$ linear in the sequence length) but it is possible to maintain a fixed number of parallel hypotheses and choose the most likely overall sequence. This is known as beam search and has cost $\mathcal{O}(BKN)$ where B is the beam width. Beam search keeps track of multiple possible sentence completions to find the overall most likely (which is not necessarily found by greedily choosing the most likely next word at

each step). Top-k sampling randomly draws the next word from only the top-K most likely possibilities to prevent the system from accidentally choosing from the long tail of low-probability tokens and leading to an unnecessary linguistic dead end.

Generative next-word estimation can be done by a Transformer decoder. GPT-3 consists of a stack of 96 Transformer decoder layers (vs. BERT's encoder-only architecture).

Each decoder layer is composed of multiple decoder blocks, which consist of masked self-attention blocks with a width of 2048 tokens and a feedforward neural network.

- The *masked self-attention* in GPT calculates the attention matrix using the **preceding** (to not cheat!) words in a sentence that can be used to estimate the next word.
- Self-attention in BERT is applied to the entire sentence.

Few-shot learning: A surprising property of large language models like GPT is that they can perform many tasks without fine-tuning! If we provide several examples of correct question/answer pairs and then another question, they often answer the final question correctly by completing the sequence.

Encoder models like BERT use only the encoder part of the original Transformer architecture (from Vaswani et al., 2017). It's designed for understanding tasks: classification, question answering, etc. It takes in a full sentence or passage at once (bidirectional), and each token attends to all others — both left and right.

Decoder models like GPT are trained to generate text by predicting the next token, given previous ones (autoregressive). Each token only attends to earlier tokens (causal masking).

Encoder-decoder models like T5 / BART are used for sequence-to-sequence tasks like machine translation, summarization, etc.

Encoders learn a representation that can be used for other tasks by predicting missing tokens. Decoders build an autoregressive model over the inputs and are an example of a generative model in this book. The generative decoders can be used to create new data examples.

- Encoder: input sequence, output single variable, e.g. sentiment analysis.
- Decoder: input single vector, output a probability distribution over values for the next token to generate a word sequence, i.e. generative models.
- Encoder-decoder: sequence-to-sequence processing tasks.

What are the Vision Transformer and ImageGPT architectures? The ViT breaks the image into a grid of patches (16×16 in the original implementation). Each of these is projected via a learned linear transformation to become a patch embedding. These patch embeddings are fed into a transformer encoder network, and the `<cls>` token is used to predict the class probabilities.

To handle 2D images, the input image is reshaped into a sequence of flattened 2D patches, after each patch is embedded into a D-dimensional vector using a trainable linear projection.

Then a constant latent vector size D is used throughout all layers. Position encodings are added to the patch embeddings to retain positional information. The resulting sequence of embedding vectors serves as input to the encoder.

The Transformer encoder in ViT consists of alternating layers of multi-headed self-attention and MLP blocks. The MLP contains two layers with GELU (Gaussian Error Linear Units) non-linearity. ViT is trained on large data sets, and fine-tuned to (smaller) downstream tasks.

ImageGPT: autoregressive image generation, and image completion.

Compare Transformer Networks to RNNs. Challenges with RNNs include long-range dependencies, gradient vanishing and explosion, requiring a large number of training steps (due to recurrence), and not being suitable for parallel computation (again, due to recurrence).

Transformer networks facilitate long-range dependencies through use of the attention mechanism. There is no gradient vanishing or explosion, fewer training steps are needed, and no recurrence relation is needed - which facilitates parallel computation.

Complexity?

The complexity of the self-attention mechanism increases quadratically with the sequence length.

Recurrent neural networks (RNNs): The word embeddings are passed sequentially through a series of identical neural networks. Each network has two outputs; one is the output embedding, and the other (orange arrows) feeds back into the next neural network, along with the next word embedding. Each output embedding contains information about the word itself and its context in the preceding sentence fragment. In principle, the final output contains information about the entire sentence and could be used to support classification tasks similarly to the `<cls>` token in a transformer encoder model. However, RNNs sometimes gradually "forget" about tokens that are further back in time.

LSTMs and GRUs partially address this. But there was also the idea that certain output words should attend more to certain input words according to their relation (Bahdanau et al., 2015). This ultimately led to dispensing with the recurrent structure and replacing it with the encoder-decoder transformer (Vaswani et al., 2017) Here input tokens attend to one another (self-attention), output tokens attend to those earlier in the sequence (masked self-attention), and output tokens also attend to the input tokens (cross-attention). Vaswani targeted translation tasks, but transformers are now more usually used to build either pure encoder (e.g. BERT) or pure decoder models (e.g. GPT). Since GPT3, many decoder language models have been released with steady improvement in few-shot results.

Attention was originally developed as an enhancement to RNNs for machine translation (Bahdanau,Section 12.2.5 Cho, and Bengio, 2014). However, Vaswani et al. (2017) later showed that significantly improved performance could be obtained by eliminating the recurrence structure and instead focusing exclusively on the attention mechanism. Today, transformers based on attention have completely superseded RNNs in almost all applications.

What are the limitations of Transformers? Two of the main limitations of Transformers are quadratic scaling and tokenisation.

The self-attention mechanism is a simple and elegant way to extract patterns from input embeddings. The source modality of these tokens (text, images, sound) and their arrival order are irrelevant. Self-attention enables effective comparison between all tokens in a set.

This differs from architectures like CNNs or RNNs, which are tailored to specific modalities. While this makes them more data efficient, the scalability of transformers often compensates: we can increase dataset size until the advantage of more biased models diminishes.

However, creating input embeddings remains highly modality-dependent. Additionally, the quadratic scaling of self-attention limits embedding granularity. Creating 10x more embeddings from the same input requires 100x more compute. Tokenization is simply a (quite brutal) way of limiting the granularity of these embeddings to individual tokens to ensure scaling. However, for text data this leads to issues like language bias and challenges in reading numbers. Similar issues exist in other modalities and even more creep up once we combine them.

The Byte Level Transformer (BLT; recent Meta paper, [19]) addresses these issues by using raw-byte data and dynamically allocating compute based on a learnable method of grouping bytes into patches. These batches don't have a fixed size and, unlike tokenization, BLT has no fixed vocabulary.

This results in a more efficient allocation of compute and holds the promise of ultimately combining different modalities at the byte level.

References

[1] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[2] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, Cambridge, 2020. ISBN 9781108455145. URL `https://mml-book.github.io/`.

[3] Mete Demircigil, Judith Heusel, Matthias Löwe, Sebastian Upgang, and Franck Vermet. On a model of associative memory with huge storage capacity. *Journal of Statistical Physics*, 168(2):288–299, 2017. doi: 10.1007/s10955-017-1806-y. URL `https://link.springer.com/article/10.1007/s10955-017-1806-y`.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2021. URL `https://arxiv.org/abs/2010.11929`. arXiv preprint arXiv:2010.11929.

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NeurIPS)*, pages 257–265. Curran Associates Inc., 2011.

[6] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018. URL `https://arxiv.org/abs/1603.07285`.

[7] K.J. Gilhooly, K. Gilhooly, F. Lyddy, and F. Pollick. *Cognitive Psychology (1st ed.)*. UK Higher Education Psychology. McGraw-Hill, 2014. ISBN 9780077122669.

[8] Varun Godbole, George E. Dahl, Justin Gilmer, Christopher J. Shallue, and Zachary Nado. Deep learning tuning playbook, 2023. URL `http://github.com/google-research/tuning_playbook`. Version 1.0.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. ISBN 9780262035613. URL `https://www.deeplearningbook.org/`.

[10] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Lecture 6e: RMSprop - Divide the gradient by a running average of its recent magnitude. Coursera Lecture: Neural Networks for Machine Learning, 2012. URL `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[11] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL `https://www.sciencedirect.com/science/article/pii/0893608089900208`.

[12] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. URL `https://arxiv.org/abs/1502.03167`.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

[14] Dmitry Krotov and John J Hopfield. Dense associative memory for pattern recognition, 2016. URL https://arxiv.org/abs/1606.01164.

[15] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks, 2015. URL https://arxiv.org/abs/1511.06807.

[16] Yurii Nesterov. A method of solving a convex programming problem with convergence rate $o(1/k^2)$. *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

[17] Andrew Ng. Machine learning. https://www.coursera.org/learn/machine-learning, 2011. Coursera, Stanford University.

[18] Christopher Olah. Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. Accessed: 2025-08-02.

[19] Artidoro Pagnoni, Ram Pasunuru, Pedro Rodriguez, John Nguyen, Benjamin Muller, Margaret Li, Chunting Zhou, Lili Yu, Jason Weston, Luke Zettlemoyer, Gargi Ghosh, Mike Lewis, Ari Holtzman, and Srinivasan Iyer. Byte latent transformer: Patches scale better than tokens, 2024. URL https://arxiv.org/abs/2412.09871.

[20] Simon J. D. Prince. *Understanding Deep Learning*. MIT Press, 2023. ISBN 9780262048644. URL https://udlbook.github.io/udlbook/.

[21] Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, Victor Greiff, David Kreil, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. Hopfield networks is all you need, 2021. URL https://arxiv.org/abs/2008.02217. Blogpost: https://ml-jku.github.io/hopfield-layers/.

[22] Jocelyn Sietsma and Robert J.F. Dow. Creating artificial neural networks that generalize. *Neural Networks*, 4(1):67–79, 1991. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(91)90033-2. URL https://www.sciencedirect.com/science/article/pii/0893608091900332.

[23] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't Decay the Learning Rate, Increase the Batch Size. *CoRR*, abs/1711.00489, 2017. URL http://arxiv.org/abs/1711.00489.

[24] Yichuan Tang and Chris Eliasmith. Deep networks for robust visual recognition. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*, pages 1055–1062, Haifa, Israel, June 2010.

[25] Matus Telgarsky. Benefits of depth in neural networks, 2016. URL https://arxiv.org/abs/1602.04485.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

[27] Wikipedia contributors. Empirical risk minimization – Wikipedia, the free encyclopedia, 2024. URL https://en.wikipedia.org/w/index.php?title=Empirical_risk_minimization. Accessed: 2025-02-20.

[28] Jong Chul Ye. *Geometry of Deep Learning: A Signal Processing Perspective*, volume 37 of *Mathematics in Industry*. Springer Singapore, 2022. ISBN 978-981-16-6045-0. doi: 10.1007/978-981-16-6046-7.

[29] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I. Jordan. How does learning rate decay help modern neural networks?, 2019. URL https://arxiv.org/abs/1908.01878.

[30] Matthew D. Zeiler. Adadelta: An adaptive learning rate method, 2012. URL https://arxiv.org/abs/1212.5701.

[31] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. https://D2L.ai.