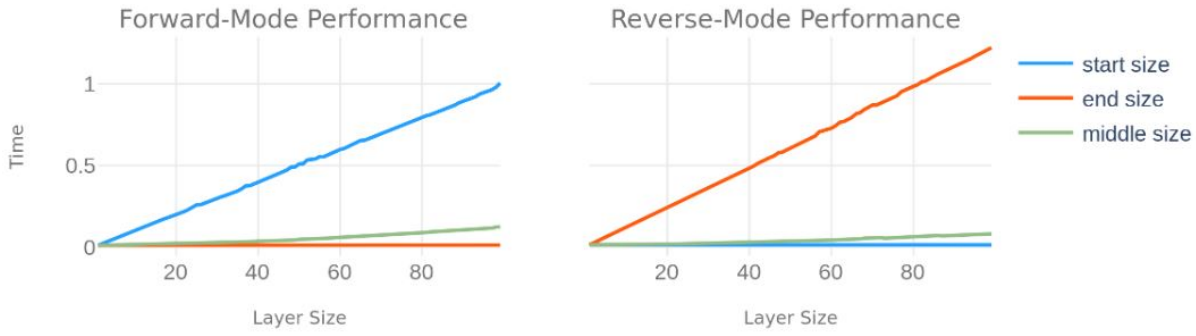
Summary Deep Learning

Linda ten Klooster

January 27, 2023

Backpropagation

The backpropagation algorithm is a direct application of dynamic programming. It contains two main phases, referred to as the forward and backward phases, respectively. During the forward phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights, resulting in a predicted outcome that is compared to the supposed value in the loss function. The main goal of the backward phase is to learn the gradient of the loss function with respect to the different weights by using the chain rule. These gradients are used to update the weights.
Backpropagation is a special case of a general technique in numerical analysis called **automatic differentiation**. **Reverse mode** automatic differentiation is used in backpropagation. **Forward mode** automatic differentiation goes more like $D_f = D_k(D_{k-1}(...(D1)...))$. While forward mode differentiation can be done in $O(1)$ memory, reverse mode differentiation requires memory roughly linear in the number of functions composed $O(k)$. The performance of both is shown below.



Loss function performance

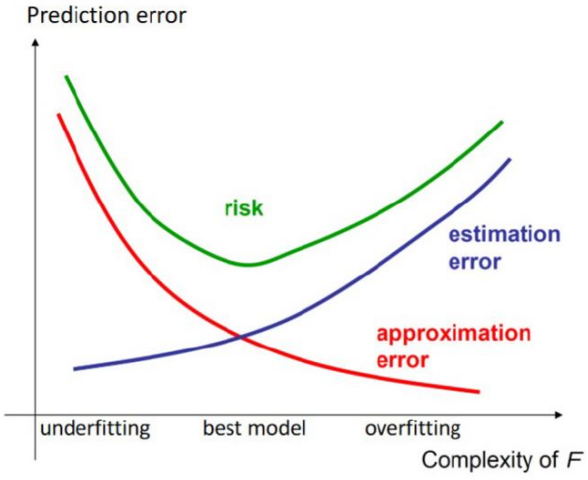
Expected risk is the **generalization performance**. How well can your model generalize to which class some new input belongs? The expected risk implies that the distribution is known.
Empirical risk is the **training performance**. It is computed directly from the data without any information about the distribution; it is an **estimator** of the expected risk.
Ideally we want to to minimize the expected risk, but not knowing the distributions, we are doomed to minimize the empirical risk.

Approximation error vs estimation error

The **approximation error** is the error implied by the choice of function class and is defined as the difference in risk obtained by the best model within the function class and the optimal model. It is the error of the model that is used.

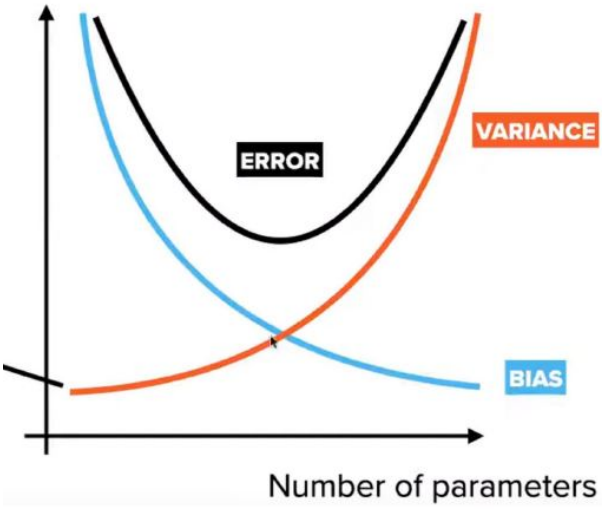
The **estimation error** is the error implied by the fact that the algorithm works with a finite training set that only partially reflects the true distribution of the data. By considering that the training set is obtained by randomly sampling the true distribution, this also incurs a certain variability in the resulting model. The estimation error captures the error towards the best estimator in the current model; it is the *difference* between the approximation error and the training error.

The cause of **overfitting** is neither using too many parameters nor using a larger model than the ground truth. But, it does occur when the decrease of approximation error is less than the increase of estimation error. When the decrease of approximation error is more significant than the increase of estimation error, this is called **underfitting**.



Bias-Variance tradeoff

The approximation error and estimation error are also related to the bias-variance tradeoff, as can be seen below.



If there are **too few parameters**, so we have an overly simple model, there will probably be a **high bias** (because it's too simplistic too capture the structure in the data) and a **low variance** (because there is not enough information to estimate the parameters, so you get the same results for any sample of the training data). In this case we have underfitting.

If there are **too many parameters**, so we have an overly complex model, there might be **low bias** (since it learns all the relevant structures) and a **high variance** (since it fits all the characteristics/features of the data you happened to sample).
The bias-variance decomposition holds only for squared error, but the trade-off provides a useful intuition even for other loss functions.

Key challenges in Neural Network optimization

- Ill-conditioning
- Local minima (non-convexity)
- Plateaus, saddle points, and other flat regions
- Cliffs and exploding gradients
- Long term dependencies (depth of architecture)
- Poor correspondence between local and global structures
- Theoretical limits of optimization

Gradient Descent

Gradient descent is an iterative optimization method, which aims to tune the parameters of the network such that the loss is minimized. The method takes very small steps in the direction opposite to the gradient to minimize the loss. Computing the gradient vector at every iteration of the steepest descent method can be costly if either the amount of parameters of the input data is very large.
It can be beneficial if the gradient of a random training point is chosen instead of the mean of the gradients of all training points. This is called **stochastic gradient descent**.
It is also possible, as a kind of intermediate solution, to take a batch of the training data to generate a gradient, this is called **batch** gradient descent.
In gradient descent the step size is called the **learning rate**, so how much we should move in a certain direction. In practice, this learning rate is often set to decay linearly until a certain iteration.

Momentum

The **momentum method** is a method to accelerate learning using stochastic gradient descent. Stochastic gradient descent suffers particularly when the error surface has a high curvature, there are small but consistent gradients, and/or the gradients are very noisy. Momentum is when we account for a change from large gradients to small gradients, think about the velocity when a ball rolls from a hill over a flat surface; it still has a lot of speed left even though the gradient is very small there. Momentum makes use of the **moving average** over past gradients.
The only problem with this method would be that it gives all past gradients the same weight. A solution for this is the **exponential moving average**, which gives the most recent gradients the highest weight.
Another approach is the **Nesterov momentum**. This method first takes a step in the direction of the accumulated gradient, then it calculates the gradient and makes a correction.

Adaptive Learning Rate Methods

Motivation: if the features vary in importance and frequency, it is probably a good idea to also vary the learning rate of the features. Here we list some different **Adaptive Learning Rate Methods**.

AdaGrad:

- Idea: downscale a model parameter by the square-root of the sum of squares of all its historical values
- Parameters that have a large partial derivative of the loss - learning rates for them are rapidly declined
- Advantage: it is good when the objective is convex
- Disadvantage: it might shrink the learning rate too aggressively, we want to keep history in mind

RMSProp

- It is an improvement on AdaGrad in non-convex settings, such that it accumulates an exponentially decaying average of the gradient
- It is also possible, and common, to use RMSProp with Nestorov
- We could have also used RMSProp with momentum, but the use of momentum with rescaling is not very well motivated

Adam

- Adam is like RMSProp with momentum but with bias correction terms for the first and second moments

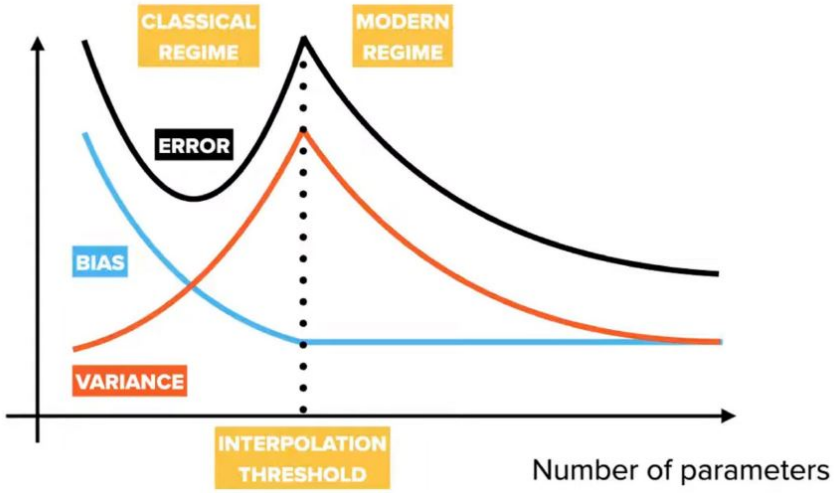
Generalization vs regularization

Generalization refers to a model's ability to adapt and react appropriately to previously unseen data chosen from the same distribution as the model's initial input. In other words, generalization assesses a model's ability to process new data and generate accurate predictions after being trained on a training set.
Regularization is a method to avoid high variance and overfitting as well as to increase generalization. Regularization aims to keep coefficients close to zero. Intuitively, it follows that the function the model represents is simpler, less unsteady. So predictions are smoother and overfitting is less likely. Below we list several regularization techniques/tricks which aim to keep both bias and variance low. The best-performing models on most benchmarks use some or all of these tricks.

- **Weight decay:** Penalize large weights, since large weights can lead to overfitting. Gradient descent updates can be seen as weight regularization, as long as a regularization term is included in the update. Examples are L1 and L2 regularization (they lead to sparsity and to more efficient neural networks).
- **Reducing the number of parameters:** One can choose to include less layers, or layers with less parameters. An alternative is a **linear bottleneck layer**. Linear layers don't make the network more expressive, but might still improve generalization.
- **Data augmentation:** This is augmenting the training data by transforming the examples. Examples are translations, flipping, rotation, smooth warping and adding noise. Do not warp the test data.
- **Equivariant layers (using symmetry of the data):** Instead of changing the data, change the network: make layers which are invariant under symmetry. An example is a CNN, which is invariant under translation.
- **Early stopping:** Monitor performance on a validation set, stop training when the validation error starts going up.
- **Implicit and iterative regularization:** Regularization by choosing a 'worse' algorithm. Balance training error vs. test error. In a non-convex landscape: global optimum of training error may not be optimal for test error. Iterative methods: Not optimal for non-convex problems, early stopping. Approximate methods: use an approximation of gradient. Stochastic methods: SGD leads to noisy approximation of gradient. Noise leads to smoothing/diffusion.
- **Ensembles (combine predictions of different models):** If you have multiple candidate models and don't know which one is the best, maybe you should just average their predictions on the test data. It often helps even when the loss is non-convex.

- **Stochastic regularization (dropout):** Regularization by injecting noise into the computations. Dropout is a stochastic regularizer which randomly deactivates a subset of the units
- **Autoencoding** is also a form of implicit regularization, because we reduce to lower dimensions
- **Variational Autoencoding** is actually regularization using KL divergence

Double descent



Convolutional Neural Networks

In general about CNNs

CNNs are scale-up NNs meant to process very large images or video sequences. They use sparse connections with a high level of parameter sharing. Not all states in a particular layer are connected to those in the previous layer in an indiscriminate way. Each layer in the convolutional network is a 3-dimensional grid structure, which has a height, width, and depth, where the depth, where the depth refers to the number of channels in each layer, for example the number of primary color channels.
The three types of layers that are commonly present in a convolutional neural network are convolution, pooling, and ReLU. In addition, a final set of layers is often fully connected and maps in an application-specific way to a set of output nodes.
CNNs are applicable to any input that is laid out on regular grids (like 1D, 2D, etc.).

Convolutional operation

In a CNN, the parameters are organized into sets of 3-dimensional structural units, known as **filters** or **kernels**. Assume that the dimensions of the filter in the qth layer are $F_q \times F_q \times d_q$. The **convolution operation** places the filter at each possible position in the image (or hidden layer) so that the filter fully overlaps with the image, and performs a dot product between the $F_q \times F_q \times d_q$ parameters in the filter and the matching grid in the input volume (with same size).
Convolutions may also use a **stride**, which is the distance between consecutive positions of the kernel. Strides are a way of downsampling within a CNN.
Some convolutional properties:

- **Sparse interactions:** a feedforward network connects every input to every output. The sparse connectivity of CNNs reduces the number of parameters significantly and improves efficiency.

- **Parameter sharing**, or tied weights, means that the CNN does not need to learn a separate set of parameters for each location, but reuses one set in every location.
- **Equivariance to translation**: due to the parameter sharing used in CNNs, the feature maps are equivariant with respect to translations. In other words, if we shifted the pixel values in the input in any direction by one unit and then applied convolution, the corresponding feature values will shift with the input values.

Padding

One observation is that the convolution operation reduces the size of a layer compared to the previous layer. This type of reduction in size is not desirable in general, because it tends to lose some information along the borders of the image. This problem can be resolved by using **padding**. The output size of the network is controlled by padding the input with zeros around its edges, effectively increasing the input size.
Different kinds of padding (with K being the kernel size):

- **'Same' padding (or half):** this adds $p = k - 1$ zeros to the input to constrain the output size to be the same for unit strides, $o = i$. For odd-sizes kernels, add $p = k/2$ zeros both sides of the input.
- **'Valid' padding** means that no padding is used.
- **'Full' padding** adds $p = 2(k - 1)$ zeros to the input (half on each side), resulting in an output size of $i + (k - 1)$.

Pooling

In typical CNNs, convolutional layers are interleaved with **pooling** layers, which compute a summary statistic over regions of the input space identified by a sliding window.
Different kinds of pooling:

- **Max pooling** computes the maximum activation per channel in the window
- **Average pooling** computes the average activation per channel in the window
- L^2 **norm** computes the 2-norm of activations per channel in the window

Key properties of CNNs

Data are compositional, they are formed of patters that are:

- **Local.** This property is inspired by visual cortex neurons. Local receptive fields activate in the presence of local features. Local connectivity means that each kernel is connected to a small region of the input image when performing the convolution. Compact support kernels. O(1) parameters per filter.
- **Stationary.** Translational invariance. Similar patches are shared across the data domain. O($n \log(n)$) parameters in general, and O(n) for compact kernels.
- **Multi-scale.** Simple structures combine to compose slightly more abstract structures, and so on, in a hierarchical way. It is inspired by the brain visual primary cortex. O(n) parameters (downsampling + pooling).

Curse of dimensionality

The difficulties related to training machine learning models due to high dimensional data is referred to as the **curse of dimensionality**. As the dimension of the feature space increases, the number of configurations can grow exponentially, and thus the number of configurations covered by an observation decreases.

Advantages of CNNs

- Solve the curse of dimensionality
- They automatically generalize across spatial translations of inputs

Disadvantages of CNNs

- Local DL: only works with regular (equidistant) spatial/temporal structures (solved by geometric DL)

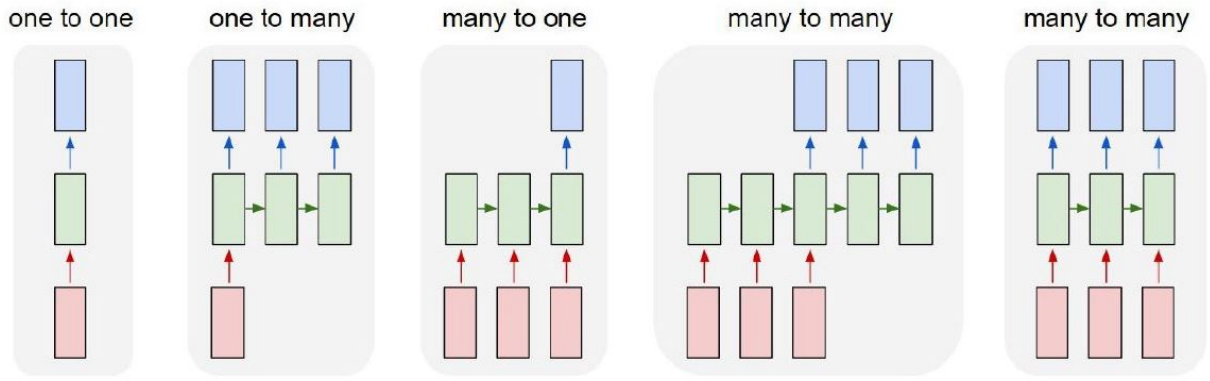
Examples of CNNs

- **AlexNet:** This network uses, amongst others, 5 convolutional layers and 3 max-pooling layers. It also makes use of ReLu activation functions, and strides of 4 in the pooling.
- **GoogLeNet:** Instead of going deeper, the layers are widened in this network. It keeps the complexity and quality of deep networks without going deep, which is an advantage because deep networks are prone to overfitting (since it's harder to pass the gradient updates through the network). An important property of GoogLeNet was that it is extremely compact in terms of the number of parameters. On the other hand, the overall architecture of GoogLeNet is computationally more expensive.
- **ResNet:** This network uses layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping. Its key feature is the use of skip connections, which create a convex loss landscape (advantage). Since we have information on both the direct and indirect root due to this skip connection, the network becomes more stable, because it overcomes the problems of vanishing and exploding gradients.

Recurrent Neural Networks

In general about RNNs

RNNs are meant for processing **sequential data**: each example consists of a pair of sequences, and most RNNs can also process examples of different lengths. Just like with CNNs, we can make use of parameter sharing across different parts of the model.
RNNs also offer flexibility, see the picture below for some different possible architectures.



Recurrence is when the a definition at time t refers back to the same definition at time $t-1$. RNNs can be built in various ways: just as any function can be considered a feedforward network, any function involving a recurrence can be considered a recurrent neural network.
We can consider the states to be the hidden units of the network. When the task is to predict the future from the past, the network learns to use the hidden state at time t, $h^{(t)}$, as a summary of task relevant aspects of the past sequence up to time t. This summary is lossy because it maps an arbitrary length sequence

$(x^{(t)}, x^{(t-1)}, ..., x^{(2)}, x^{(1)})$ to a fixed vector $h^{(t)}$. Depending on the training criterion, the summary might selectively keep some aspects of the past sequence with more precision. The most demanding situation for $h^{(t)}$ is to approximately recover the input sequence.

Challenge of Long-term Dependencies & Long Short-Term Memory (LSTM)

The basic problem of learning long-term dependencies is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization). In the recursive method the eigenvalues of W are raised to the power of t, and so quickly there is either a decay to zero or an explosion (during backpropagation). This problem is particular to RNNs. The solution to this is either Residual RNNs and Neural ODEs.
A solution to this problem is to change the recurrence equation for the hidden vector with the use of the LSTM with the use of long-term memory. The operations of the LSTM are designed to have fine-grained control over the data written into this long-term memory.
The LSTM is an enhancement of the RNN architecture in which we change the recurrence conditions of how the hidden states are propagated. In order to achieve this goal, we have an additional hidden vector which is referred to as the cell state. One can view the cell state as a kind of long-term memory that retains at least a part of the information in earlier states by using a combination of partial "forgetting" and "increment" operations on the previous cell states. The upshot of this phenomenon is that the resulting neural network is able to model long-range dependencies. This is achieved by using a gentle approach to update these cell states over time, so that there is greater persistence in information storage. Persistence in state values avoids the kind of instability that occurs in the case of the vanishing and exploding gradient problems. One way of understanding this intuitively is that if the states in different temporal layers share a greater level of similarity (through long-term memory), it is harder for the gradients with respect to the incoming weights to be drastically different.
The Gated Recurrent Unit (GRU) is even better than LSTM. Here an additional variable is introduced, and you can say you stabilize by averaging. It can be viewed as a simplification of the LSTM, which does not use explicit cell states. Another difference is that the LSTM directly controls the amount of information changed in the hidden state using separate forget and output gates.
Two ways to deal with overcoming the long-term dependencies are as follows

- **Skip Connections:** Add connections from the distant past to the present. For Vanilla RNNS: the recurrence goes from a unit at time t to a unit at time $t+1$. Normally, the gradients vanish/explode with respect to the number of time steps. With recurrent connections with a time delay of d, gradients explode/vanish exponentially as a function of $\frac{\tau}{d}$ rather than τ.
- **Leaky Units:** Keep a running average for a hidden unit by adding a linear self connection: $h_t \leftarrow \alpha h_{t-1} + (1-\alpha)h_t$. This ensures hidden units can easily access values from the past.

Residual RNNs

Residual RNNs are a real game changer. It are RNNs with residual blocks added. If you study ResNets you can see it as a forward Euler discretization of an ODE. After analyzing this we can say that the ODE is stable if

- The Jacobian changes slowly in time
- $Re(eig\mathbf{K(t)}) \leq 0$ for every t

Now since we learn $\mathbf{K}$ (the parameters), stability is ensured by regularization/constraints.

Neural ODE

The goal of Neural ODEs is a continuous-time/depth model via differential equations and numerical integration solvers. Its key idea is that reverse mode differentiation (backpropagation) is classical adjoint state method. Advantages computational properties

- **Memory savings:** ODE backwards via ODE forward with reverse time → Memory cost constant in number of evaluations of f. In ResNet memory cost grows linearly.
- **Adaptivity:** Predicted trajectories can be automatically compared with extrapolations → Adaptive ODE solvers (>100 years knowledge) achieve this by using limited resources.
- **Speed/precision:** Controllable trade off of speed/precision at train and test time via error tolerance in ODE solvers → More flexible approach than weight pruning (sparsification) or quantization.

Disadvantages computational properties

- **Speed:** ODE Nets will generally require more inner layer evaluations than a fixed architecture on same task → Jacobian or kinetic regularization might fix it.
- **Hyperparameters:** Extra hyperparameters in solver → Could be seen as a feature.

Advantages modeling properties

- **Tractable change of variables:** discrete to continuous limit simplifies calculations, e.g. in normalizing flows.
- **Continuous-time time series models:** Dealing with irregularly sampled data.
- **Smooth homeomorphisms:** For example PointNet can fit 3D surfaces to data

Disadvantages modeling properties

- **Restrictions on activation functions**
- **Deterministic dynamics:** Extension to stochastic ODEs need but possible.

Advantages of RNNs

- **Parameter sharing** allows to share statistical strength and generalize to lengths of sequences not seen during training
- The model can be applied to **sequences of different lengths** not seen during training

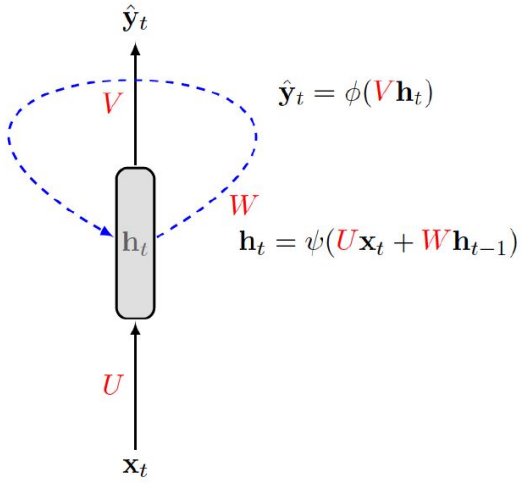
Disadvantages of RNNs

- RNNs are **very hard to train** because of the fact that the time-layered network is a very deep network, especially if the input sequence is long. In other words, the depth of the temporal layering is input-dependent.
- A big challenge associated with an RNN is that of the **vanishing and exploding gradient problems**.
- Even though the loss function has highly varying gradients to the variables in different layers, the same parameter matrices are shared by different temporal layers. This combination of varying sensitivity and shared parameters in different layers can lead to some unusually unstable effects

Examples of RNNs

- **Vanilla RNN:** It's the most basic RNN, not a very good one yet. It produces an output at each time stamp and has recurrent connections between hidden units. It is also Turing Complete. It has input and output sequences of the same length. The architecture can be found below.

  A disadvantage is that both the forward and propagation take time O(t), so it cannot be parallelized. The state at time t only captures information from the past.



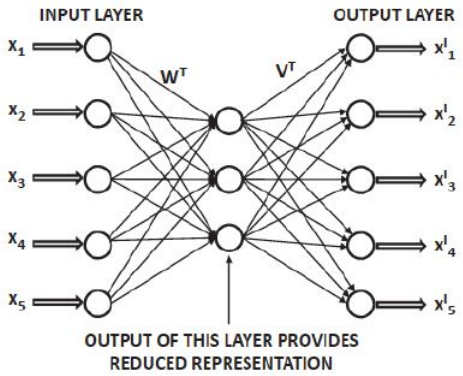
ψ can be tanh and ϕ can be softmax

- **Bidirectional RNN:** The output $y(t)$ may depend on the whole input sequence in this case. The bidirectional RNN looks both into the future and the past to disambiguate interpretations (i.e. don't just look at a sentence (text/sound) word by word). An application is speech recognition, or handwriting recognition.
- **Encoder-decoder:** This RNN deals with mapping input sequences to output sequences of different lengths. The input is the context C of which we want to find some representation. The architecture is done such that the input sequence is encoded, which this is then decoded by the decoder and makes a sequence again. It finds a common underlying state that represents the sequence/recursion. An application of encoder-decoder is text translation, or question answering.

Autoencoders

In general about AEs

The goal of using unsupervised learning via AEs is getting meaningful features that capture the main factors of variation in the dataset. These are good for classification, clustering, exploration, generation,... We have no ground truth for them.
The basic idea of an AE is to have an output layer with the same dimensionality as the inputs. The idea is to try to reconstruct each dimension exactly by passing it through the network. An AE replicates the data from the input to the output, and although this might seem like a trivial matter by simply copying the data forward from one layer to another, this is not possible when the number of units in the middle are constricted. In other words, the number of units in each middle layer is typically fewer than that in the input/output. As a result, these units hold a reduced representation of the data, and the final layer can no longer reconstruct the data exactly. Therefore, this type of reconstruction is inherently lossy. The loss function of this neural network uses the sum-of-squared differences between the input and the output in order to force the output to be as similar as possible to the input.
The figure below shows that **Singular Value Decomposition (SVD)** is a special case of an AE architecture. SVD is a dimensionality reduction method $D \approx UV^T$. The n rows of D contain the n training points, the n rows of U provide the reduced representations of the training points, and the k columns of V contain the orthogonal basis vectors. The rows of the matrix D are the input to the encoder, the activations of hidden layers are rows of U and the weights of the decoder contain V. The reconstructed data contain the rows of UV^T.
If we use the MSE as the loss function, this becomes the same objective function as SVD. It is possible for gradient-descent to arrive at an optimal solution in which the columns of each of U and V might not be



mutually orthogonal. However, the subspace spanned by the columns of each of U and V will always be the same as that found by the optimal solution of SVD.
The optimal encoder weight matrix W will be the pseudo-inverse of the decoder weight matrix V if the training data spans the full dimensionality. Tying encoder-decoder weights does not lead to orthogonality for other architectures, but it is a common practice anyway.
Low dimensional latent space forces the network to learn a "smart" compression of the data, not just an identity function. The encoder and decoder can have arbitrary architecture.
Linear transformations for the encoder and decoder give results close to PCA (Principal Component Analysis).
Deeper networks give better reconstructions, since the basis can be non-linear.

Deep AEs

Deep AEs work so well because deep NNs have the potential to linearize, even though the network itself uses a nonlinear map. Better reductions are obtained by using increased depth and non-linearity.

Linear transformations for the encoder and decoder give results close to PCA (Principal Component Analysis).
Deeper networks give better reconstructions, since the basis can be non-linear.

Semi-Supervised Classification

Assume we have many images, but few ground truth labels. Then we start with unsupervised learning: train the AE on many images. Afterwards we then do some supervised fine-tuning: we train the classification network on labeled images.

Generative Models

The assumption is that the dataset are samples from an unknown distribution $p_{data}(x)$. The goal is then to create a new sample from $p_{data}(x)$ that is not in the dataset.
How can we measure the similarity of $p_{data}(x)$ and p_θ, which is sampled from the probability distribution from the latent space, where θ are our parameters?

- Likelihood of data in $p_\theta(x)$ (**VAEs**)
- Adversarial game: The discriminator distinguishes $p_{data}(x)$ and p_θ vs. Generator makes is hard to distinguish (**GANs**)

Both of these methods are based on the idea that you have a latent code, but there is an element of randomness in this code.
In autoencoders we have a trained decoder which transforms some features z to approximate samples from $p_{data}(x)$. The generative model randomly selects a dataset from the whole class, which then goes through the latent space to be decoded.

Examples of AEs

- **Variational AEs:** Probabilistic spin to traditional autoencoders → allows generating data. It defines an intractable density → derive and optimizes a (variational) lower bound.
  It trains the generator to maximize the likelihood of the data in $p_\theta(x)$.
  To generate a sample from the model, the VAE first draws a sample from the code distribution. The sample is then run through a differentiable generator network. Finally, x is sampled from a distribution model. During training, however, the approximate inference network (or encoder) is used to obtain the sample, and the probability distribution of the model is then viewed as a decoder network.
  Advantages are:
  1)Principled approach to generative models
  2)Allows inference of q(z—x), can be useful feature representation for other tasks
  3) The VAE framework is straightforward to extend to a wide range of model architectures
  Disadvantages are:
  1) Maximizes lower bound of likelihood: this is okay, but not as good an evaluation as PixelRNN/PixelCNN
  2) Samples blurrier and lower quality compared to state-of-the-art (GANs)

Generative Adversarial Networks

In general about GANs

GANs don't work with an explicit density function. Instead, they work with two neural network models simultaneously. The first is a generative model that produces synthetic examples of objects that are similar to a real repository of examples. Furthermore, the goal is to create synthetic objects that are so realistic that it is impossible for a trained observer to distinguish whether a particular object belongs to the original data set, or whether it was generated synthetically.
The second network is a discriminative network that has been trained on a data set which is labeled with the fact of whether the images are synthetic or fake. The discriminative model takes in inputs of either real examples from the base data or synthetic objects created by the generator network, and tries to discern as to whether the objects are real or fake.
In a sense, one can view the generative network as a "counterfeiter" trying to produce fake notes, and the discriminative network as the "police" who is trying to catch the counterfeiter producing fake notes. Therefore, the two networks are adversaries, and training makes both adversaries better, until an equilibrium is reached between them.

Formal formulation

We have a generator G and a discriminator D.
G, with parameters θ_G, takes a noisy r.v. z as an input (with a noise prior distribution $p_z(z)$) and generates candidate samples matching the data distribution x. So the mapping to the data space is $G(z, \theta_G)$.
D, with parameters θ_D, takes a sample data point as an input and outputs a single scalar $D(x, \theta_D)$. $D(x)$ represents the probability that x came from the data, rather than the generator-implied distribution p_G.
We train D to maximize the probability of assigning the correct (binary) label both to training examples and samples from G.
We train G to maximize $\log[D(G(z))]$, or, equivalently, minimize $\log[1 - D(G(z))]$.
Like so, training becomes a min-max game: $\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{x \tilde{} p_x}[\log D(x)] + \mathbb{E}_{z \tilde{} p_z}[\log(1 - D(G(z)))]$.

Challenges

- **Mode collapse.** This seems to arise as a direct consequence of the way the adversarial loss is defined. When many input noise values are mapped to the same datapoint, the generator lacks diversity in the samples it can generate. Primarily it could be seen as an issue of under-fitting. The problem of mode collapse has been studied in various contexts.

- **Vanishing gradients.** This is related to the assumption of infinite capacity of the models. This happens because the data distribution and the model distribution are in a non-overlapping lower dimensional manifold. In such a case the discriminator can be trained to perfection leading to vanishing gradients. Under the assumption of infinite capacity of the models, the Jensen-Shannon divergence saturates.
- **Non-convergence.** While performing simultaneous gradient descent for training the generator and the discriminator, it is tricky to decide upon the number of iterations for which the generator and discriminator have to be trained. Improper balance of the number of iteration results in *Undamped Oscillations*. Ideally training the discriminator to optimality between every generator updates not only is computationally expensive but also results in a pessimistic discriminator leading to the problem of vanishing gradients. The convergence of the GAN game holds true if the function is convex, the model has infinite capacity, and enough training samples are available. These assumptions are not valid in practice. Non-convergence of the simultaneous gradient descent has been discussed in many papers. The convergence is not guaranteed sometimes even in convex cases like $xy = 0$.
- **Existence of equilibrium and generalization.** The GAN game is modeled as a two-player zero-sum game on continuous strategy space. Guarantees on the existence of Unique Pure strategy Nash Equilibrium in a continuous strategy space is also an important question to be asked.
- **Evaluation of generative models.** The foremost challenge is that there is no precise and unique quantitative or qualitative measure for evaluating the generated images. The generative images of the original paper use Parzen Window estimates of the model's log-likelihood. The evaluation is also compared based on the model's performance on some surrogate tasks like classification, de-noising or missing value imputation. The most widely accepted measure, for now, is the Inception score. Ultimately, it is desirable to have a metric that evaluates both diversity and visual fidelity simultaneously.

Advantages of GANs

- Beautiful, state-of-the-art samples
- **High quality samples** from high dimensional distributions

Disadvantages of GANs

- They are trickier/more **unstable** to train (optimization problem trickier than for VAEs)
- They can't solve inference queries such as $p(x), p(z|x)$
- Deep Convolutional GANs have **sensitive hyperparameters:** use of batchnorm, strided convolutions, careful learning rates, several updates D per G updates

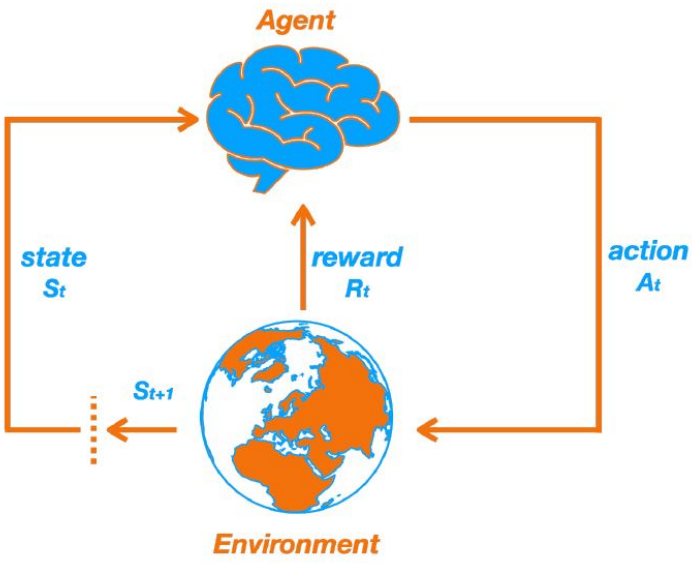
Examples of GANs

- **Conditional GANs:** The generator no longer makes use of Gaussian noise z, but it rather is conditioned by an input x (e.g. image labels, black and white color). Mode collapse would probably happen less here.
- **Cyclic GANs:** No alignment between pairs needed, simply two different sets of images.
- **Bidirectional GANs:** Here, GANs meet autoencoders. The network learns to generate and discriminate, but also learn how to encode. The complexity of the optimization becomes very high however.
- **Wasserstein GANs:** When the distribution p_θ does not exist (e.g. when data distributions are supported by low dimensional manifolds), the Kullback-Leibner distance is not defined $\rightarrow$ use Wasserstein distance instead. This uses some concepts from Optimal Transport. 'Regular' GANs cannot recover from a poor initialization (without overlap between supports of real data distribution and the generator's). The advantage of Wasserstein GANs is that they do allow generation from poor initialization data.

Deep Reinforcement Learning

General framework

Reinforcement learning (RL) is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must **discover which actions yield the most reward by trying them**. This is called learning by **trial and error**.



The agent is a **decision-making entity**, where the environment can be unknown, linear, stochastic,etc. The goal of the agent is learning to select actions to perform the task optimally. Optimality in RL is the maximization of the total cumulative future reward.

Challenges of RL

- Actions may have long term consequences
- Reward may be delayed
- It may be better to sacrifice immediate rewards to gain more long-term reward
- Not all the relevant information from the world may be available
- No model or prior knowledge assumed available
- Exploration vs exploitation (hard to do both at the same time)
- Time matters, i.e. sequential data, not i.i.d.

Agent-Environment Interaction

There is a discrete-time interaction. At each time step, the agent observes the environment through an **observation** O_t, acts through an **action** A_t and thus changes the environment, receives a **reward** R_t assessing the quality of the action taken from the environment, and receives a new observation O_{t+1} from the environment.
The environment state S_t^e is the environment's private representation, e.g. the set of numbers, equations the environment uses to determine what happens next. This state is not always accessible by the agent and it may contain irrelevant information for taking optimal decisions.

The agent state S_t^a is the agent's internal representation, so the information the agent uses to select the action. The agent's state is a function of the history $S_t^a = f(H(t))$, with $H(t) = O_0, R_0, A_0, ..., O_t, R_t, A_t$.
S_t^e and H_t are both **Markov**, so it contains all the useful information from the history, but the future is independent of the past given the present.
Full observability: the agent directly observes the environment state: $O_t = S_t^e$. Due to full observability, S_t^e is included in the history and therefore, the best choice the agent can make for its state is $S_t^a = S_t^e = O_t$.
If we have a Partially Observable Markov Decision Process (POMDP), the agent has to build its own state representation S_t^a. For this it can for example use the complete history, the belief of the environment state, or a recurrent NN.

Markov Decision Process

Most of the problems we face in RL can be formalized as MDPs. A policy π fully characterizes the behavior of an agent: $\pi(a|s) = p(A_t = a|S_t = s)$. MPD policies depend on the current state and they are time-independent (stationary). They can be either deterministic or stochastic.
The value function $\nu_\pi(s)$ gives the long-term value, or the expected return, of the state s when the policy is followed by the agent. It evaluates the goodness/badness of the state.
It can be mathematically convenient to discount rewards, since we then avoid infinite returns (in infinite-horizon MDPs). However, uncertainties about the future may not be fully represented.
The value function can be written as the sum of the immediate reward R_t and the discounted value of the successor state $\gamma\nu_\pi(S_{t+1})$.
The(state-)action-value function $q_\pi(s, a)$ gives the long-term value of the state s, taking action a when the policy π is followed by the agent. It evaluates the goodness/badness of taking action a in state s. This one can also be decomposed into the immediate reward and the discounted value of the successor state.
The optimal value function $\nu_*(s)$ specifies the best possible performance in the MDP. This function satisfies the so-called Bellman Equation, and solving the Bellman Equation for all states and actions results in finding the optimal value function. However, all these evaluations require the transition model between states and the reward model, and it is possibly unfeasible for MDPs with a high number of states and actions.

Exploration vs Exploitation

Since the agent does not know the optimal solution, it must try new policies (thus **exploration**) without losing too much rewards (thus **exploitation**). This is a dilemma, since we need both.

Taxonomy of RL

- **Online RL** learn the value function/policy while interacting with the environment
- **Offline RL** learn the optimal value function/policy from a dataset collected beforehand and only afterwards interact with the environment
- **Off-policy** algorithms use target policy (policy that we want to learn, thus exploit), and behavior policy (policy used to generate the date, thus explore)
- **On-policy** algorithms use a unique policy that either explores or exploits

Value-based methods estimate the optimal value function and implicitly derive the optimal policy from it:

- **Monte Carlo (MC):** uses the experience of the complete episode. The idea is that the value is the mean (expected) return of a certain policy
- **Temporal Difference (TD):** uses only one (or a few) step ahead experience. It updates a guess with a guess, i.e. bootstrapping

Q-learning

Q-learning is an off-policy TD learning method that can learn optimal long-term behaviors without any knowledge of the model of the environment.
For an appropriate choice of a(learning rate/step size): $Q(s,a)$ converges to $q_*(s,a)$ and its greedy policy $\pi(s) = \text{argmax}_a Q(s,a)$ converges to an optimal policy $\pi_*(s)$

Value function approximation

Value functions are represented by **look-up tables**. Every state and every state-action pair have an entry. However, what happens in large MDPs? Here we have to problems:

- It **cannot store** all the entries **in memory**
- It is **unfeasible to evaluate** the value of each state (or state-action pair) individually, i.e. loop-up tables have no generalization capabilities

This calls for a need for **approximating the value functions** via function approximators (e.g. neural networks).
The advantages of this are:

- **Generalization** from seen states (state-action pairs) to unseen states (state-action pairs)
- **Same learning principles:** parameters can be learning with MC or TD-learning

This also means we need **differentiable** function approximators, for example neural networks, decision trees, linear combination of features, etc. We also need training methods that are suitable for **non-stationary and non-i.i.d. data**.

Deep Q-Network

Deep Q-network is the first successful example of Deep Reinforcement Learning algorithm. It extends Q-learning to non-linear function approximators.
However, there are some challenges to make DQN work well in practice:

- **Reuse of experience** (i.e. data gathered from the interaction with the environment)

  The solution is **Experience Replay**: store all the data in a memory buffer such that we can reuse them multiple times for updating our value function
- **Temporal correlation** of the data (i.e. data from trajectories)

  The solution is **sampling random minibatches** from the memory buffer. Minibatches are composed of non-temporally correlated data
- **Instability** of the Q-learning update (i.e. use of the same network for predicting the target value function)

  The solution is to use a **target network** to generate a **fixed target** (not updated directly)

Challenges of Deep RL

- **Sample efficiency:** Deep RL algorithms require huge amount of data
- **Training stability:** algorithms easily diverge
- **Learning from limited data**
- **Generalization:** learning policies on an environment and transfer to new ones without further retraining
- **Choice of the reward function:** hard to define a proper reward function for complex tasks
- **Exploration in complex environments:** random actions do not guarantee sufficient exploration
- **Interpretability:** better understanding of how the agents select actions from the observations

Graph Neural Networks

We can **extend template matching for graphs**. There are however two main issues with this:

- **No node ordering:** How to match template features with data features when nodes have no given position or order (index is not a position)?
- **Heterogeneous neighborhood:** How to deal with different neighborhood sizes within a graph?

Vanilla Graph Convolutional Networks

This is the simplest formulation of spatial GCNs. Below some properties are listed.

- They handle the absence of node ordering → Invariant by node re-parametrization
- They deal with different neighborhood sizes
- Local reception field by design (only neighbors are considered)
- Weight sharing (convolution property)
- Independent of graph size
- Limited to isotropic capability → You deal with every vertex in the same way

Graph Attention Networks (GAT)

GAT uses the **attention mechanism** to introduce anisotropy in the neighborhood aggregation function. The network employs a **multi-headed architecture** to increase the learning capability, similar to the Transformer. Transformers is a special case of GCNs when the graph is fully connected. It doesn't make sense to talk about a graphs in this case, since the whole graph is the neighborhood of a node, so **Transformers are Set NNs**.