Summary Deep Learning

Phillip Lippe

November 1, 2020

Contents

1 Introduction

1.0.1 Perceptron

- Single perceptron weights every element of the input with a weight, and adds a bias term
- Step function as output: if input sum greater zero, then output is 1, else 0 (or -1)
- Problem: can only learn linear problems and not e.g. XOR
- Overcoming by multi-layer perceptron; however, Rosenblatt's algorithm not applicable because learning depends on ground truth, which does not exist for intermediate neurons

1.0.2 Deep Learning today

- now, better hardware, bigger data
- makes sense if raw data is uninterpretable $\rightarrow$ either create representation or learn
- while highly non-convex, hypothesis is that most local minima are close to global minimum

1.0.3 Linear separability

- let $(x, l)_n : x_i \in \mathbb{R}^d$, then there are $M = 2^n$ possible datasets if target is binary and all x constant
- only (about) d out of M are linearly separable and probability of linear separability decreases very fast when $n > d$
- Solution: have non-linear features that are invariant (but not too invariant), repeatable (but not bursty), discriminative (but not too class-specific), robust (but sensitive enough) $\rightarrow$ learn features from data
- Hypothesis: raw data live in huge dimensionalities but effectively lie in lower-dimensional non-linear manifolds $\rightarrow$ discover these manifolds

2 Modular Learning

- *Definition*: A family of parametric, non-linear and hierarchical representation learning functions, which are massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.
- A neural network is a series of hierarchically connected functions $\Rightarrow$ Directed Acyclic graph
- Note that it is not allowed to have loops except over time/additional dimension

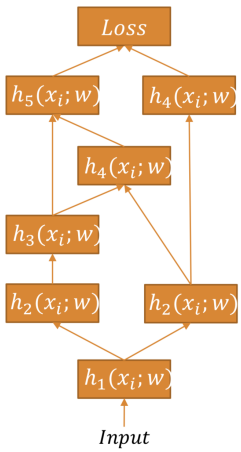


Figure 1: Example network with interweaved connections. The architecture can be made arbitrarily complex, and can also include recurrent connections.

2.1 Module

- A module is the simplest mathematical component in a NN, and can be expressed by $a = h(x; w)$ where a is the output, x the input, w trainable parameters (optional) and h an activation function (must be 1st-order differentiable (almost) everywhere)
- Use not too complex models, better complex hierarchies
- trade-off between model complexity and efficiency (often more training iterations with weak model better than few iterations with strong model)
- w mostly learned by gradient-based methods, usually maximizing the likelihood
  - ML solution: $w^* = \arg\max_w \prod_{x,y} p_{model}(y|x; w)$
  - For gradient-based methods, we can minimize the negative log likelihood: $\mathcal{L}(w) = -\mathbb{E}_{x,y\sim\hat{p}_{data}}[\log p_{model}(y|x; w)]$
  - If output is Gaussian, we would get the ℓ_2 norm
  - If output is Laplacian, we would get the ℓ_1 norm
- Using a loss function that matches the output distribution of the network helps, because:
  - It makes math simpler (exponential cancels out)
  - Better numerical stability ($\log$ with very small/negative values, helps for e.g. Softmax+CrossEntropy)
  - Makes gradients larger as exponential-like activations often lead to saturation, which means gradients are almost 0 (but not with $\log$)
- It is important that the input and output distribution of every module match, as otherwise we get inconsistent behavior and makes it harder to learn
  - For activation functions, this means we prefer them to be mostly activated around the origin and centered
  - Otherwise, e.g. ReLU can be come a linear unit or set everything to 0

2.1.1 Example modules

- **Linear module**: $h(x; w) = xw^T + b$
  - Simple gradients $\frac{\partial h}{\partial w} = x$, $\frac{\partial h}{\partial x} = w$
  - No activation saturation $\Rightarrow$ strong, reliable gradients
- **Rectified Linear Unit**: $h(x) = \max(0, x)$
  - Gradient is step function. $\frac{\partial h}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$
  - Hence, strong, fast gradients; ReLUs are data efficient
  - However, dead neurons might be an issue when initialization/weights produce outputs smaller 0 for every input; also, large gradients might cause a neuron to die (higher learning rates can help)
    - Leaky ReLU: $h(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{if } x \leq 0 \end{cases}$; $\frac{\partial h}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ a & \text{if } x \leq 0 \end{cases}$
    - ELU: $h(x) = \begin{cases} x & \text{if } x > 0 \\ \exp(x) - 1 & \text{if } x \leq 0 \end{cases}$; $\frac{\partial h}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ \exp(x) & \text{if } x \leq 0 \end{cases}$
    - Swish: $h(x) = x * \sigma(x)$; $\frac{\partial h}{\partial x} = \sigma(x)(1 + x - x\sigma(x))$
    - Swish: $h(x) = \ln(1 + e^x)$; $\frac{\partial h}{\partial x} = \frac{1}{1 + e^{-x}}$
- **Sigmoid**: $h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$
  - Gradient easy to calculate: $\frac{\partial h}{\partial x} = \sigma(x)(1 - \sigma(x))$
  - Can be used as output function for probability distribution between $[0, 1]$
  - Saturates (easily becomes overconfident i.e. 0 or 1 decisions) and has small gradients

  - Not centered around origin ⇒ not good choice for within a network

- **Tanh**: $h(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Gradients $\frac{\partial h}{\partial x} = 1 - \tanh(x)^2$
  - Saturates as well, but has slightly higher gradients than sigmoid and is centered around origin → less "positive" bias for next layers
- **Softmax**: $h(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
  - Probability distribution over multiple classes
  - Softmax trick for numerical stability: $\frac{e^{x_i - \mu}}{\sum_j e^{x_j - \mu}}$, where $\mu = \max_i x_i$
- **Euclidean Loss**: $h(x, y) = 0.5||y - x||^2$
  - suitable for regression problems
  - sensitive to outliers (magnifies errors quadratically)
- **Cross Entropy Loss**: $h(x, y) = -\sum_{j=1}^{K} y_j \log x_j$, where $y_j \in \{0, 1\}$
  - suitable for classification problems
  - derived from max likelihood learning
  - couples well with softmax/sigmoid

2.2 Backpropagation

- Calculate gradients of all parameters in the network based on the loss on the last layer
- Principle of chain rule: $\frac{\partial z}{\partial x} = \sum_i \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$ (gradients from all possible paths)
  - **TODO** In vector notation: $\nabla_{\boldsymbol{x}} \boldsymbol{z} = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \cdot \nabla_{\boldsymbol{y}} \boldsymbol{z}$ with Jacobian $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix}$
- Steps of Backpropagation:
  1. Compute forward propagations for all layers recursively: $\mathbf{h}_l = h_l(\mathbf{w}; \mathbf{x}_l)$ and $x_{l+1} := \mathbf{h}_l$
  2. Compute the reverse path. **TODO**

$$\frac{\partial \mathcal{L}}{\partial a^{(l)}} = \left(\frac{\partial a^{(l+1)}}{\partial x^{(l+1)}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a^{(l+1)}}, \quad \frac{\partial \mathcal{L}}{\partial \theta^{(l)}} = \frac{\partial a^{(l)}}{\partial \theta^{(l)}} \cdot \left(\frac{\partial \mathcal{L}}{\partial a^{(l)}}\right)^T$$

  3. Use gradients $\frac{\partial \mathcal{L}}{\partial \theta^{(l)}}$ to update parameters via SGD

3 Deep Learning Optimizations

- Pure optimization has a very direct goal, namely finding the optimum. However, in Machine Learning, we want to not only make small errors on training data but also generalize well to new data. Thus, the "optimal" parameters might not necessarily be the optimum (e.g. overfitting)

3.1 Stochastic Gradient Descent

- Pushing the weights towards highest gradient change

$$w_{t+1} = w_t - \eta_t \nabla_w \mathcal{L}$$

- *(Batch) Gradient descent*: gradients on the full dataset. However:
  - Dataset is mostly too large for this
  - No real guarantee that this leads to a good optimum and/or it will converge faster
- *Stochastic gradient descent*: approximate gradients from random mini-batch.

  - Standard error is inverse proportional to number of elements m in a batch: $\sigma/\sqrt{m}$, where σ is variance in $p(x, y^*)$
  - Noisy gradients help to escape local minima, acts as regularization, reduce overfitting (good shuffling important)
  - Does sample roughly representative gradients from dataset. Is better as training data is also just a rough approximation of what the test data might look like (optimum on training $\neq$ optimum on test)
  - SGD is faster, especially in first iterations
  - SGD is able to adapt with dynamically changing datasets (and also good for streaming data)
  - make sure of class/data balance in batches
- *Ill conditioning*: if gradients are large, applying them can lead to worse performance. This is the case if the second order derivative changes faster

	Batch GD	Stochastic GD
Conditions of convergence	Yes	No
Apply Hessians & accelerations on curvatures	Yes	No
Theoretical analysis	Yes	No
Scales up/efficient	No	Yes
Guaranteed theoretical fast convergence	No	No
Overfitting	Easier	Harder
Global minimum	No	No
In practice, good results	No	Yes

Figure 2: SGD vs GD

3.2 Advanced optimizations

3.2.1 Gradient-based optimization

- *Pathological curvatures*: move through a ravine towards minimum. SGD tends to oscillate between the walls because they have high gradients
  - Second order optimization can help a lot for pathological curvatures:

$$w_{t+1} = w_t - H_{\mathcal{L}}^{-1} \eta_t g_t$$

  - Hessian $H_{\mathcal{L}}^{ij} = \frac{\partial \mathcal{L}}{\partial w_i \partial w_j}$ works as adaptive learning rate per parameter
  - However, unfeasible in practice because Hessian gets very large

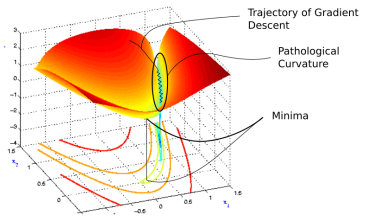


Figure 3: Pathological curvature

- **Momentum**: maintain *momentum* from previous parameter updates to dampen the oscillations.

$$u_{t+1} = \gamma u_t - \eta_t g_t$$
$$w_{t+1} = w_t + u_{t+1}$$

  - Works as a exponential averaging $\Rightarrow$ more robust gradients, faster convergence
  - γ might be initialized lower and then increased over time to 0.9

  - Standard values for γ are between 0.5 and 0.9 (note that a lower learning rate should be used compared to standard SGD)

- **RMSprop**: adapting learning rate on current loss surface.

$$r_t = \alpha \cdot r_{t-1} + (1-\alpha) \cdot g_t^2$$
$$\eta_t = \frac{\eta}{\sqrt{r_t} + \epsilon}$$
$$w_{t+1} = w_t - \eta_t \cdot g_t$$

  - r_t is the (exponentially) averaged gradient norm describing the size of the gradients (per dimension!)
  - The learning rate is then adapted by η_t at every time step for each dimension independently
  - ϵ to prevent numerical instability and too large learning rates
  - With the adapted learning rate, we update our weights with SGD

- **Adam**: Combining adaptive learning rate and momentum

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1) \cdot g^{(t)}$$
$$v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2) \cdot \left(g^{(t)}\right)^2$$
$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1^t}, \hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^t}$$
$$w^{(t)} = w^{(t-1)} - \frac{\eta}{\sqrt{v^{(t)}} + \epsilon} \circ \hat{m}^{(t)}$$

  - Keeps track of the gradient norm for momentum $m^{(t)}$, and norm (also known as velocity) $v^{(t)}$
  - The hyperparameters β_1 and β_2 correlate with the γ and α respectively from the previous approaches
  - The adaptive learning rate is expressed by $\hat{v}^{(t)}$, and the exponentially averaged gradients by $\hat{m}^{(t)}$
  - The division is to remove the bias of $m^{(0)}$ and $v^{(0)}$ being zero. Note that β_1^t means the value of β_1 to the power t, and not at time step t
  - Adam is in general better for complex models, but might fail on easy/stupid tasks compared to simple methods like SGD

- **Adagrad**: adapting learning rate based on both gradient scale and frequency of updates

$$G_t = G_{t-1} + \text{diag}\left(g_t^2\right)$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

  - Very similar to RMSprop, but sums the scales over all time steps (G_t) instead of exponentially averaging
  - Less sensitive to learning rate tuning, but it gets very small over training time annealing to 0

- **Nesterov momentum**: use the future gradient instead of the current gradient. Leads to better convergence in theory

3.2.2 Bayesian optimization

- Gradient-based optimizations have the problem of getting stuck in local minima
- Bayesian optimization is a gradient-free, educated trial and error guesser that works in lower dimensional spaces (up to 1000, but mostly 20 to 50 parameters)
- Determines the next point/parameter values to evaluate based on variance/uncertainty, and expected/predictive value.
- Can be used for e.g. network architecture search

3.3 Normalization

- Data pre-processing
  - Center data around 0 (activation functions are designed for that)
  - Scale input variables to have similar diagonal covariances (not if features are differently important)
  - De-correlate features if there is no inductive bias (e.g. sequence over time)
- **Batch normalization**: ensure Gaussian distribution of features over batches at every module input

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i, \quad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$
$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma^2 + \epsilon}}$$
$$\hat{y}_i = \gamma \cdot \hat{x}_i + \beta$$

  - Normalize feature to $\hat{x}_i \sim \mathcal{N}(0, 1)$, then rescale with trainable parameters γ (variance) and β (mean).
  - Helps the optimizer to control mean and variance of input distribution, and reduces effects of 2nd order between layers $\Rightarrow$ easier, faster learning
  - Acts as regularizer as distribution depends on mini-batch and therefore introduces noise
  - During testing, take a moving average of the last training steps and use those for μ_B and σ_B^2

3.4 Regularization

- Weight regularization needed to prevent overfitting
- ℓ_2**-regularization**: Introduce objective term for minimizing weights

$$w^* = \arg\min_w \mathcal{L} + \frac{\lambda}{2}\sum_l ||w_l||^2$$

  - When using simple (stochastic) gradient descend, then ℓ_2 regularization is the same as weight decay:

$$w_{t+1} = (1 - \lambda\eta_t)\, w_t - \eta_t \nabla_\theta \mathcal{L}$$

- ℓ_1**-regularization**: use ℓ_1 objective, introduces sparse weights

$$w^* = \arg\min_w \mathcal{L} + \lambda\sum_l ||w_l||$$

- **Early stopping**: stop the training when test error increases but training loss continues to decrease. Can be counted to regularization as training steps are reduced
- **Dropout**: setting activations randomly to 0 during training with probability p (mostly between 0.1 and 0.5)
  - During test time, every activation is reweighted by $1 - p$
  - Reduces co-adaptations/-dependencies between neurons because none can solely depend on the other
  - Neurons get more robust $\Rightarrow$ reduces overfitting
  - Effectively, a different network architecture is used every iteration. Testing can be seen as using model ensemble

3.5 Weight initialization

- There are two forces on the weight magnitude: small weights are needed to keep data around origin, but large weights are required to have strong learning signals
- Initialization should preserve variance of activations (input variance $\approx$ output variance to keep distribution between modules same)

- Depends on non-linearity and data normalization
- **Xavier initialization**: to maintain data variance, the variance of the weights must be $1/d$ where d is number of input neurons $\Rightarrow$ sample weight values from $w \sim \mathcal{N}(0, \sqrt{1/d})$
- **Initialization for ReLU**: ReLU set half of the output neurons to 0 $\Rightarrow$ double the weight variance to compensate zero flat-area: $w \sim \mathcal{N}(0, \sqrt{2/d})$

4 Convolutional Neural Networks

- Images are stationary signals with spatial structure and huge dimensionality
- Input dimensions are highly correlated (e.g. translation invariant)
- Preserve spatial structure by convolutional filters, local connectivity (with shared weights) and being robust to local variances by spatial pooling

4.1 Transfer Learning

- Use large datasets like ImageNet to learn useful features for other, smaller datasets
- Prevent overfitting, even for large networks
- Alternatively, we could also use a pre-trained network on task 1 as feature extractor for task 2 (same as freezing first layers)
- Which layer(s) to fine-tune?
  - If both task have the same labels, we can initialize all layers. Otherwise, the classification layer (last layer) must be newly trained. If there is only very few data available, only fine-tune this layer
  - If datasets are very different, the fully connected layers need to be replaced
  - First convolutional filters capture low-level information that mostly does not change over datasets. Mid-level convolutions can be fine-tuned if dataset is large enough
- Use a smaller learning rate for pre-initialized layers as network starts already from a point close to the optimum. New layers can be trained with higher learning rate

4.2 Standard classification architectures

4.2.1 VGGNet

- All filter sizes are 3×3, as this is the smallest filter size, and is more parameter efficient to build up large filters, plus additional non-linearity between filters
- 1×1 convolutions used to increase non-linearity/complexity without increasing receptive field

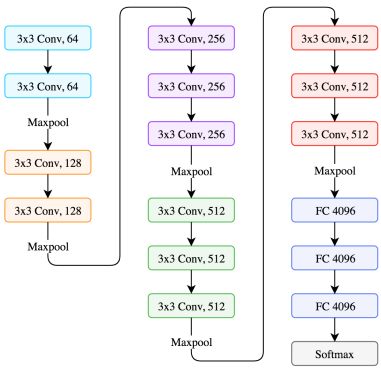
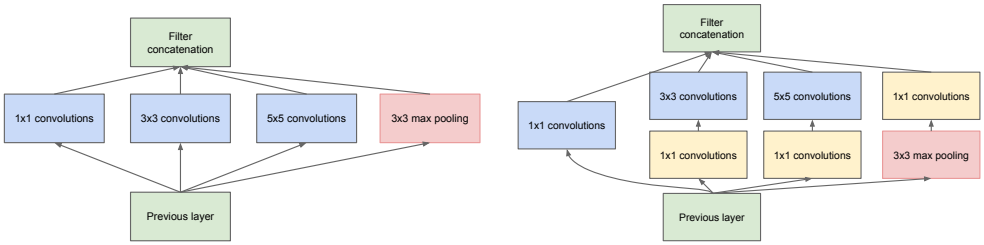


Figure 4: VGG16 architecture

4.2.2 Inception

- Receptive fields should vary in size as objects can appear in different scales
- Naively stacking more convolutional operations on top of each other is expensive and prone to overfitting
- Inception module applies different filter sizes on same input (1×1 convolutions for feature reduction)
- Architecture consists of 9 Inception blocks
- Solution for vanishing gradients: have intermediate classifiers that amplify the gradient signal for early layers
- InceptionV2: 5×5 replaced by two 3×3 filters
- InceptionV3: 1×3 and 3×1 filters instead of 3×3
- BatchNormalization has shown to be very helpful in this architecture

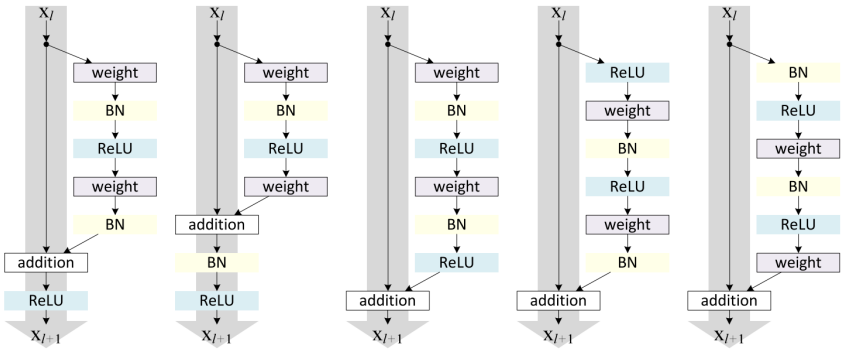


(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 5: Inception module

4.2.3 ResNet/DenseNet/HighwayNet

- Deeper networks are harder to optimize, and might actually achieve worse results than shallow ones because of that (although learning identity in additional layers must lead to same results)
- Better approach: try to model the difference that is learned in every layer $H(x) = F(x) + x$
- Different ways for modeling $F(x)$. Most popular ones shown in Figure 6. BatchNormalization has been shown to be very important because of vanishing gradients



(a) original (b) BN after addition (c) ReLU before addition (d) ReLU-only pre-activation (e) **full pre-activation**

Figure 6: ResNet blocks

- **HighwayNet** introduces a gate with learnable parameters to determine the importance of a layer: $H(x) = F(x) \cdot T(x) + x \cdot (1 - T(x))$
- **DenseNet** uses skip connections to multiple forward layers. Creates complex blocks where last layer sees the input of all previous layers

4.3 Tracking/Object detection

4.3.1 Fast R-CNN

- Based on middle feature map, get bounding boxes by e.g. selective search
- RoI pooling returns fixed size feature map for selected bounding box (puts e.g. 3×3 mask on features and pools accordingly)
- Features used to generate class prediction and location correction
- During training, sample multiple candidate boxes from image and train on all of them. Makes it more efficient/faster, *but* batch elements might be highly correlated (in the paper, they report that they experienced it to be neglectable)
- Very accurate and fast, but external box proposals needed
- **Faster R-CNN**: train network to propose box locations

4.3.2 Siamese Network for Training

- Use Siamese network to compare similarity of two patches
- If we compare patches over time, we can find objects with the highest similarity $\Rightarrow$ tracking of objects
- Can be trained on rich video dataset, and can be applied to unseen categories/targets

4.4 Spatial Transformer Network

- ConvNets must be invariant/robust to pose/geometry changes. One simple way of doing it is data augmentation
- Better: use spatial transformer network to learn rotation/scale transformation
- Define grid on input. Scale, translation and rotation parameters are learned by the network and depend on the input. Finally, transform image based on the changed grid.
- Operation is differentiable and thus can be learned

5 Recurrent and Graph Neural Networks

5.1 Backpropagation through time

- Sequences are of arbitrary length. Standard networks like CNN mostly work on fixed input dimensionality
- Usage of memory with shared weights θ:

$$c_{t+1} = h_\theta\left(x_{t+1}, c_t\right) = h_\theta\left(x_{t+1}, h_\theta\left(x_t, c_{t-1}\right)\right) = ...$$

- Simple RNN cell:

$$\begin{aligned} c_t &= \tanh\left(U \cdot x_t + W \cdot c_{t-1}\right) \\ y_t &= \text{softmax}\left(V \cdot c_t\right) \\ \mathcal{L} &= \sum_{t=1}^{T} y_t^* \log y_t \end{aligned}$$

- Gradient for output weights V:

$$\begin{aligned} \frac{\partial \mathcal{L}_t}{\partial V} &= \frac{\partial \mathcal{L}_t}{\partial y_t}\frac{\partial y_t}{\partial c_t}\frac{\partial c_t}{\partial V} = \left(y_t - y_t^*\right) \cdot \left(c_t\right)^T \\ \frac{\partial \mathcal{L}}{\partial V} &= \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial V} \end{aligned}$$

- Gradient for memory weights W:

$$\frac{\partial \mathcal{L}_t}{\partial W} = \frac{\partial \mathcal{L}_t}{\partial y_t} \frac{\partial y_t}{\partial c_t} \frac{\partial c_t}{\partial W}$$

  - In $\frac{\partial c_t}{\partial W}$, c_t depends on c_{t-1} which again depends on W. Thus, we have a recurrence in the gradient calculation:

$$\frac{\partial \mathcal{L}_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}_t}{\partial y_t} \frac{\partial y_t}{\partial c_t} \frac{\partial c_t}{\partial c_k} \frac{\partial c_k}{\partial W}$$

    where $\frac{\partial c_k}{\partial W}$ only models the dependency exactly at time step k
  - The gradient $\frac{\partial c_t}{\partial c_k}$ can be determined by the chain rule: $\frac{\partial c_t}{\partial c_k} = \prod_{i=k+1}^{t} \frac{\partial c_i}{\partial c_{i-1}}$
  - All in all, the final loss is:

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \mathcal{L}_t}{\partial y_t} \frac{\partial y_t}{\partial c_t} \left(\prod_{i=k+1}^{t} \frac{\partial c_i}{\partial c_{i-1}} \right) \frac{\partial c_k}{\partial W}$$

- Gradient for input weights U very similar to W:

$$\frac{\partial \mathcal{L}}{\partial U} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \mathcal{L}_t}{\partial y_t} \frac{\partial y_t}{\partial c_t} \left(\prod_{i=k+1}^{t} \frac{\partial c_i}{\partial c_{i-1}} \right) \frac{\partial c_k}{\partial U}$$

- The problem with RNNs are that the gradients at time step t depend on c_{t-1} which also depends on w. However, the gradients are calculated with the assumption that w stays the same for the previous time steps.
- This error can easily accumulate over many time steps so that in very long sequences, the gradients for the last steps are inaccurate
- Reduce learning rate/fewer updates, but this leads to slower training

5.1.1 Vanishing gradients

- The exact derivations can be found in this paper
- We assume an alternative formulation for simplicity here: $c_t = W \cdot \sigma(c_{t-1}) + U \cdot x_{t-1}$ where σ is an arbitrary activation function. Then, the partial derivative between two time steps is
  $\frac{\partial c_t}{\partial c_k} = \prod_{i=k+1}^{t} \frac{\partial c_i}{\partial c_{i-1}} = \prod_{i=k+1}^{t} W^T \cdot \text{diag}\left(\frac{\partial \sigma(c_t)}{\partial c_t}\right)$
- Hence, the magnitude of $\frac{\partial c_{t+1}}{\partial c_t}$ is bounded by this derivative:

$$\left\| \frac{\partial c_{t+1}}{\partial c_t} \right\| \leq \left\| W^T \right\| \cdot \left\| \text{diag}\left(\frac{\partial \sigma\left(c_t\right)}{\partial c_t} \right) \right\|$$

- In case the derivative of our non-linearity is bounded to a value γ (which is 1 in case of tanh), we know that gradients vanish if the norm of the weight gradients are lower than $1/\gamma$:

$$\left\| \frac{\partial c_{t+1}}{\partial c_t} \right\| \leq \left\| W^T \right\| \cdot \left\| \text{diag}\left(\frac{\partial \sigma\left(c_t\right)}{\partial c_t} \right) \right\| < \frac{1}{\gamma} \gamma = 1$$

- This term is exponentiated with the number of time steps. Thus, long sequences suffer even more of vanishing gradients $\Rightarrow$ learn only short-term relationships
- If however $\left\| \frac{\partial c_{t+1}}{\partial c_t} \right\| > 1$ because of $\left\| W^T \right\| \gg 1/\gamma$, then we can get exploding gradients
- Quick fix for exploding gradients: clip gradient norm. However, there the counterpart can happen where we only focus on long-term relationships

5.1.2 Long Short-Term Memory

- Preventing vanishing gradients by gate mechanism
- By simply adding features to memory and limiting memory by sigmoid we can get strong gradients for any sequence length. Note that the gradients get lower in expectation because sigmoid has mean 0.5. Nevertheless, if long-term dependencies are important, the network can learn them now
- *Forget gate*: regulating how much information is kept from last time step
- *Input + candidate gate*: Regulating which, and how much new information should be added given the current time step
- *Output gate*: What features are important for the current time step

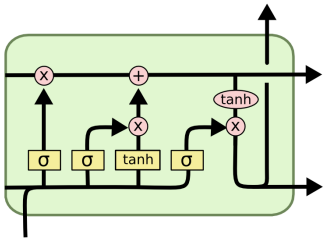


Figure 7: Visualization of a LSTM cell

5.2 Graph Neural Networks

- Perform operation on graph-structured data (e.g. social networks or knowledge graphs)

5.2.1 Deep Walk

- Learning latent representations of vertices in a network
- The Deep Walk algorithm consists of two simple steps:
  1. Perform random walks on the graph to generate node sequences
  2. Run skip-gram on sequence (with word window) to learn node embeddings
- *Drawback*: algorithm has to be re-run if a new node is added, not useful for dynamic graphs

5.2.2 GraphSage

- In every iteration, aggregate information of neighbors and the node itself to generate new embeddings
- Aggregation techniques are taking the mean (with weight and non-linearity applied on it afterwards), max pooling, or using a LSTM

5.2.3 Graph Convolutional Networks

- A GNN layer takes as input the embeddings for every node $H^{(l)}$ and the adjacency matrix A, and create new embeddings $H^{(l+1)}$
- Graph convolutional layers use for this a matrix multiplication where weights are shared over nodes
- In the simplest form, a GCN layer can be defined as $h(H^{(l)}, A) = \sigma\left(AH^{(l)}W^{(l)}\right)$
- To make it more efficient, we add the identity matrix to $\hat{A} = A + I$ so that nodes use their old embeddings as well, and take the mean instead of the sum over all neighbors (by degree matrix D):

$$h(H^{(l)}, A) = \sigma\left(D^{-1/2}\hat{A}D^{-1/2}H^{(l)}W^{(l)}\right)$$

6 Deep Generative Models

- *Generative modeling*: learn the joint probability $p(x, y)$ or density function $p(x)$. Task can be performed with Bayes rule: $p(y|x)$. Generalize better (less prone to overfitting), and better modeling of causal relations. Members include GAN, VAE, etc.
  - We can use generative models to predict uncertainty and out of distribution examples: $p(x, y) = p(y|x)p(x) \Rightarrow$ if x o.o.d., then $p(x)$ low!
- *Discriminative modeling*: learn conditional pdf $p(y|x)$. Is usually task-oriented and gets better results.
- Applications of generative models
  - Simulating possible futures for reinforcement learning
  - Creating missing data (e.g. pixel patches which are missing)
  - Super-resolution scaling for images
  - Data augmentation (replace e.g. car by bicyclist in a scene)
  - Cross-modal translation (sketch to image)
- Different type of generative models (see Figure 8)
  - *Explicit density*: maximize log likelihood of the data by modeling a probability density function. Function must be complex enough and computationally tractable
  - *Implicit density*: no explicit pdf needed, only a sampling mechanism

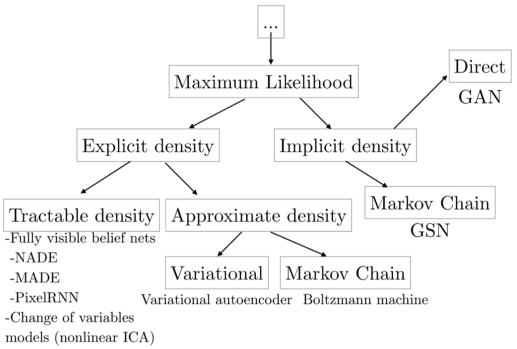


Figure 8: Overview of generative models

6.1 Generative Adversarial Networks

- Adversarial training of generator vs discriminator

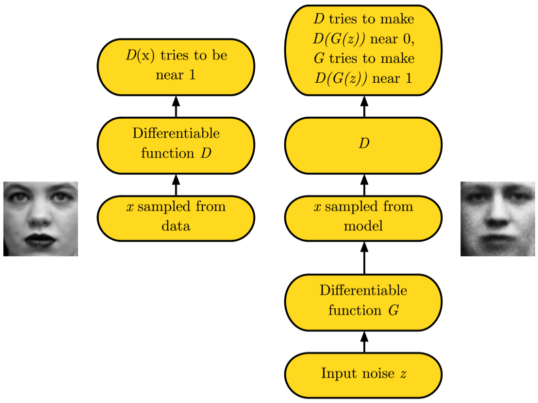


Figure 9: Pipeline of adversarial GAN training

- The generator is a (mostly deconvolutional) network that takes noise z as input, and creates fake images. The discriminator tries to distinguish between fake and real images

- Trained in a minimax game fashion, the loss function resembles the Jensen-Shannon divergence:

$$\min_G \max_D V(G,D) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}\left[\log\left(D\left(\boldsymbol{x}\right)\right)\right] + \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}\left[\log\left(1 - D\left(G\left(\boldsymbol{z}\right)\right)\right)\right]$$

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] - \frac{1}{2}\mathbb{E}_{z\sim p_z}\left[\log 1 - D(G(z))\right]$$

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_{z\sim p_z}\left[\log D(G(z))\right]$$

- Loss of generator is changed from $\log 1 - D(G(z))$ because otherwise the gradients of the generator vanish for a too strong discriminator
- Divergence is important and can strongly influence the behavior of model

$$D_{KL}\left(p(x)\|q^*(x)\right) = \int p(x)\log\frac{p(x)}{q^*(x)}dx \implies \text{if } p(x) > 0, \text{ then } q(x) > 0$$

$$D_{KL}\left(q^*(x)\|p(x)\right) = \int q^*(x)\log\frac{q^*(x)}{p(x)}dx \implies \text{if } p(x) = 0, \text{ then } q(x) = 0$$

6.1.1 GAN training problems

- **Vanishing gradients** during training:
  - If the discriminator is too bad, the generator does not get valid/accurate feedback and can therefore not learn properly
  - If the discriminator is perfect, the generator has very low gradients as a small change does not influence the discriminator

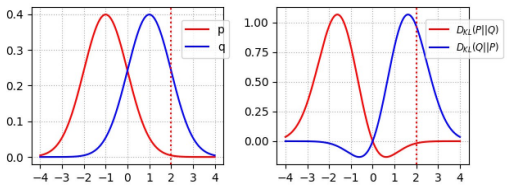


Figure 10: Vanishing gradients problem for training with KL-divergence. When the distance between the two distributions p and q (respectively P_g and P_r) is too huge, the KL divergence is very close to zero. Hence, is does not provide any strong gradients in these regions.

- **Reaching the equilibrium**
  - We know that the nash equilibrium of the minimax game is $P_g = P_r$ meaning the distribution of the real data is equal to the generated data. In that case, D return 0.5 no matter what example we put in (as both distributions are equal).
  - However, it has been shown that such cost functions may not converge when using gradient descent. An example is shown in Figure 11.

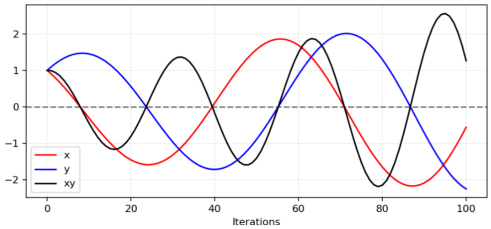


Figure 11: Oscillating behavior of a non-cooperative game where $\min_x \max_y V(x,y) = x \cdot y$. The equilibrium $x = y = 0$ is never reached.

- **Mode collapse**
  - A GAN suffers from a mode collapse if the generator limits its predictions/generated distribution to a few samples/modes.

- For example in case of the MNIST dataset, this would mean that the generator only creates numbers of one or two different digits. Although a full mode collapse is rarely the case, partial mode collapses frequently occur
- In order to create a mode collapse, the gradients regarding the noise $\boldsymbol{z}$ must be very low/close to zero. This can for example happen if we fix the discriminator and the generator converges to the optimal image $\boldsymbol{x}^*$ that fools the discriminator the most
- Once the generator collapse to one mode, the discriminator will learn that this mode is purely/mostly generated and thus changes its predictions. The generator will address that by changing the mode (note that as $\partial L / \partial \boldsymbol{z} \approx 0$, we will just collapse to the next mode and are not able to escape this loop).
- In the end, this turns into a cat-and-mouse game between the generator and discriminator, and will not converge (see Figure 12).

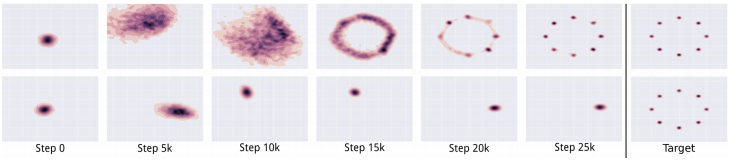


Figure 12: *Top row*: optimal convergence of generator distribution to 8 modes. *Bottom row*: Sample of a mode collapse after 10k iterations. The generator is only able to generate a single mode.

- **Low dimensional support**
  - The KL and JS divergence work best for overlapping distributions as neither of them is 0 (numerical instability)
  - However, during training, the training distribution is not perfect, and as we have high dimensional data, both distributions are less likely to overlap much
  - Also, it is easy for the discriminator to find a line in between them

6.1.2 GAN improvements

- **Wasserstein GAN**
  - Instead of KL/JS, use Wasserstein (Earth Mover's) Distance:

$$\mathcal{W}(p_r, p_g) = \inf_{\gamma \sim \prod(p_r, p_g)} \mathbb{E}_{(x,y)\sim\gamma} |x - y|$$

  - Intuitive explanation: how much do I have to move from one distribution to get the other one. Thus, the distance is even meaningful for non-overlapping distributions
- **Usage of labels**
  - Learning a conditional model $p(y|x)$ often generates better samples than from a random distribution
  - One example are conditional GANs where we have given a ground truth
- **Label smoothing**
  - Train the discriminator to predict $D(x) \approx 1 - \alpha$ instead of 1
  - Has been shown to be a good regularization by preventing the discriminator to be overconfident
  - In addition, the gradients of the generator do less likely explode
- **Virtual batch normalization**
  - Batch Normalization can significantly help in neural networks
  - However, in GANs, it leads to high intra-batch correlation
  - Solution: *virtual batch normalization* where we select a reference batch which is fixed during training, and combine it with the statistics of the current batch. Reduces overfitting on reference batch and intra-batch correlation

6.1.3 GAN open questions

- **Mode collapse**: How to prevent a model to suffer from mode collapse. One idea is penalizing the model is features are too similar, or allowing discriminator to see across batch elements. But these solutions are more heuristic tries and no theoretical solution
- **Evaluation of GANs**: GANs are currently judged by their qualitative results/predictions, but there is no quantitative measurement yet
- **Discrete outputs**: The generator and discriminator need to be differentiable, and thus discrete outputs are not possible. There are some workarounds, but no real theoretically sound solution.
- **Semi-supervised classification**: How to combine a GAN training and discriminative model efficiently (discriminator predicts class and fake/real at the same time)

6.2 Boltzmann machines

- A Boltzmann distribution is defined by $p(x) = \frac{1}{Z}\exp\left(-E\left(x\right)\right)$ where $E(x)$ is a energy function described by our model, and $Z = \sum_x \exp\left(E\left(x\right)\right)$ a normalization constant
- The benefit of defining a distribution like that is that our model can use any output values between $[-\infty, \infty]$ instead of being constrained to $[0, 1]$
- A problem is that even if x is binary, the normalizing constant Z gets out of hands (sum over 2^n combinations for n dimensional x). Thus, we limit the computations by only considering pairwise relations
- Pairwise relations modeled by $E(x) = -x^T W x - b^T x$. Learning W and b by maximizing the likelihood of the data
- Problem: W is still of size n^2 which can be too large for e.g. images (256×256 leads to 4.2 billion parameters in W) $\Rightarrow$ Restricted Boltzmann machines

6.2.1 Restricted Boltzmann machines

- Restrict model by additional bottleneck over h latents

$$E(x,h) = -x^T W h - b^T x - c^T h, \;\; p(x) = \frac{1}{Z}\sum_h \exp\left(-E\left(x,h\right)\right)$$

- This function is not in the form of a energy function anymore (because of the sum). We can rewrite it as:

$$\begin{aligned} F(x) &= -b^T x - \sum_i \log \sum_{h_i} \exp\left(h_i\left(c_i + W_i x\right)\right) \\ p(x) &= \frac{1}{Z}\exp\left(-F(x)\right) \\ Z &= \sum_x \exp\left(-F(x)\right) \end{aligned}$$

- Can be represented as a single MLP layer (undirected) with less hidden units
- Compared to simple Boltzmann machine, we can express higher-order relations
- Every hidden unit is independent of each other, and the same for input x:

$$p(h|x) = \prod_j p(h_j|x,\theta), \;\; p(x|h) = \prod_i p(x_i|h,\theta)$$

- We can now reformulate the conditional probabilities as sigmoids **iff** h and x are still binary:

$$p(h_j|x,\theta) = \sigma\left(W_{:,j}x + b_j\right), \;\; p(x_i|h,\theta) = \sigma\left(W_{i,:}h + c_i\right)$$

- The loss is maximizing the log likelihood:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_n \log p(x_n|\theta) = \frac{1}{N}\sum_n \left[-F(x) - \log Z\right]$$

- The gradients can be computed accordingly:

$$\frac{\partial \log p(x_n|\theta)}{\partial \theta} = -\sum_h p(h|x_n,\theta)\frac{\partial E(x_n,h|\theta)}{\partial \theta} + \sum_{\tilde{x},h} p(\tilde{x},h|\theta)\frac{\partial E(\tilde{x},h|\theta)}{\partial \theta}$$

  Problem: second term is sum over x and $h \Rightarrow$ high-dimensional, hard to compute

- One way to do it is using contrastive divergence: sample $h_0 \sim p(h|x)$, and $x_1 \sim p(x|h_0)$, etc. In practice, a single sample is mostly sufficient

- **Deep Belief Network**: RBM are still models of single layer, we can also use a stack of RBMs. First layer is directed, others not. Our joint pdf is $p(x,h_1,h_2) = p(x|h_1)\cdot p(h_1|h_2)$

- **Deep Boltzmann machines**: also a stack of RBMs, but with undirected first layer
  - Hence, we get $p(h_2^k|h_1,h_3) = \sigma\left(W_1^{:,k}h_1 + W_3^{k,:}h_3\right)$
  - Computing gradients is intractable $\Rightarrow$ approximate by sampling

6.3 Variational Autoencoders

- We assume an underlying, lower-dimensional data distribution $p(z)$ with which we can model our data distribution $p(x,z) = p(x|z)p(z)$

- Therefore, we need to model $p(z|x)$ which is often not easy to compute. In variational inference, we approximate the true posterior by $q_\varphi(z)$ (approximated posterior does not have to depend on observed x, e.g. in VAE it does)

- Our goal is to maximize $p(x)$. As this is intractable, we use the ELBO:

$$\begin{aligned}
\log p(x) &= \log \int p(x,z)dz \\
&= \log \int q_\varphi(z)\frac{\int p(x,z)}{q_\phi(z)}dz \\
&= \log \mathbb{E}_{q_\varphi(z)}\left[\frac{p(x,z)}{q_\varphi(z)}\right] \\
&\geq \mathbb{E}_{q_\varphi(z)}\left[\log \frac{p(x,z)}{q_\varphi(z)}\right] \\
&= \mathbb{E}_{q_\varphi(z)}\left[\log p(x|z)\right] - \mathrm{KL}\left(q_\varphi(z)||p(z)\right) = \mathrm{ELBO}_{\theta,\varphi}\left(x\right)
\end{aligned}$$

- The distance between $\log p(x)$ and the ELBO is the KL divergence to the true (unknown) posterior:

$$\log p(x) - \mathrm{KL}\left(q_\varphi(z)||p(z|x)\right) = \mathbb{E}_{q_\varphi(z)}\left[\log p(x|z)\right] - \mathrm{KL}\left(q_\varphi(z)||p(z)\right)$$

- Thus, maximizing the ELBO either increases the log likelihood or optimizes the approximated posterior

- Variational Autoencoders make $q_\varphi(z)$ dependent of x, and model $p_\theta(x|z)$ as well:

$$\mathrm{ELBO}_{\theta,\varphi}\left(x\right) = \mathbb{E}_{q_\varphi(z|x)}\left[\log p_\theta(x|z)\right] - \mathrm{KL}\left(q_\varphi(z|x)||p_\lambda(z)\right)$$

  Note that $p_\lambda(z)$ is not optimized, and its parameters λ just describe the prior (e.g. standard Gaussian)

- The loss function for a VAE is the negative ELBO, where we approximate the expectation by a single sample. The KL is mostly chosen to be analytically solvable (e.g. for two Gaussian) to prevent a Monte-Carlo approximation of the integral

- However, we face a problem when we try to compute the gradients for $\nabla_\varphi \mathcal{L}$. Using Monte-Carlo integration has high variance, and sampling is non-continuous operation

- **Reparameterization trick**: sample from external, constant distribution, and transform this sample into a sample of the modeled distribution. For Gaussian: $z = \mu_q + \sigma_q \cdot \epsilon$

6.3.1 Improvements of VAE

- **Encoder distribution**
  - Modeling $q(z|x)$ as Gaussian makes training and implementation easy, but assumes that true posterior is also Gaussian, or can be at least approximated by one
  - Simple option: use different task-specific distribution like e.g. hyperspherical, however not always suitable
  - We can improve the complexity of this posterior by plugging in a Normalizing flow on top of the encoder output

$$z_0 \sim q_0(z|x) = \mathcal{N}(z|\mu(x), \text{diag}(\sigma^2(x)))$$
$$q_K(z|x) = q_0(z|x) \cdot \left|\det \frac{\partial f_K(z_{k-1})}{\partial z_{k-1}}\right|$$

  - The ELBO is added with an additional term during training

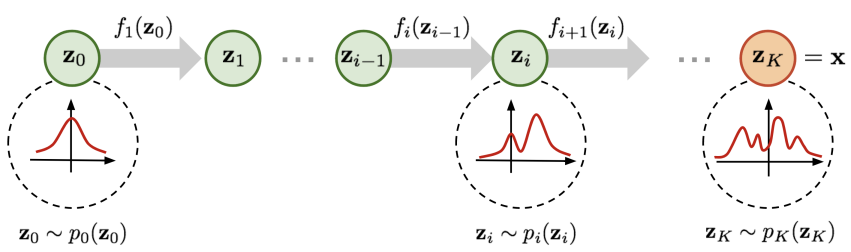
$$\text{ELBO} = \mathbb{E}_{q_\varphi(z|x)}\left[\log p_\theta(x|z)\right] - \text{KL}\left(q_\varphi(z|x)||p_\lambda(z)\right) + \mathbb{E}_{z_0 \sim q_0(z_0|x)}\left[\sum_{k=1}^{K} \log \left|\det \frac{\partial f_k(z_{k-1})}{\partial z_{k-1}}\right|\right]$$

- **Prior optimization**
  - We assume a prior $p(z)$ which is for example Gaussian, but cannot make sure that every point of the prior actually has a realistic counterpart in the original x space
  - The optimal prior is the averaged distribution over all data samples: $q^*(z) = \frac{1}{N}\sum_{n=1}^{N} q_\varphi(z|x_n)$
  - However, summing over all data point is infeasible. Thus, approximate it by K pseudo-inputs u_k that are trained via standard SGD in the framework:

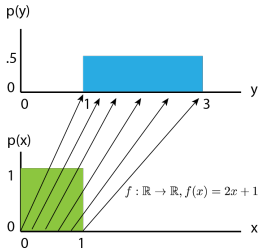
$$p_\lambda(z) = \frac{1}{K}\sum_{k=1}^{K} q_\varphi(z|u_k)$$

6.4 Normalizing flows

- VAE cannot model $p(x)$ directly because of the intractable formulation ($p(x) = \int p(x,z)dz$)
- Normalizing Flows solve that problem by using a series of invertible transformation that allow more complex latent distributions than Gaussian
- The models can therefore be trained on directly maximizing the log likelihood instead of using the ELBO or similar
- A normalizing flow consists of multiple flows that transform a simple Gaussian distribution step by step in the data distribution (see Figure 13a)
- Every flow shifts the probability mass specified by parameters (determined by e.g. a NN, see Figure 13b)



(a) General concept of stacking multiple flows



(b) Shifting density

Figure 13: Outline of how a normalizing flow works

- Mathematically, we can define a normalizing flow by:

$$x = z_k = f_k \circ f_{k-1} \circ ... \circ f_1(z_0) \rightarrow z_i = f_i(z_{i-1})$$

$$p(z_i) = p(z_{i-1}) \cdot \left| \det \frac{f_i^{-1}}{z_i} \right| \implies p(x) = p(z_0) \cdot \prod_{i=1}^{K} \left| \det \frac{f_i^{-1}}{z_i} \right|$$

$$\log p(x) = \log p(z_0) - \sum_{i=1}^{K} \log \left| \det \frac{f_i}{z_i} \right|$$

- Requirements: f must be invertible (dimensions of x and z equal), and the Jacobian must be easy to compute (i.e. triangular)

7 Bayesian Deep Learning

- Bayesian machine learning: holding a distribution per latent variable instead of single value
- Benefits of Bayesian
  - Ensemble modeling (better accuracies)
  - Uncertainty estimates, preventing overconfident networks
  - Model compression (have prior that pushes weights towards 0)
  - **Think of more**

7.1 Epistemic uncertainty

- *Epistemic uncertainty*: dataset limits
- Uncertainty that is introduced by dataset limits (unseen data $\Rightarrow$ how certain are the weights)
- Can be reduced by increasing the amount of data
- Important for safety-critical applications and small datasets
- Hard to model because posterior is usually intractable for complex functions like NN

$$p(w|x,y) = \frac{p(x,y|w)p(w)}{\int p(x,y|w)p(w)dw}$$

- **Monte-Carlo Dropout**: apply dropout during testing (Bernoulli-distribution over weights as variational distribution). The variance/uncertainty derived from there approximates uncertainty gained by variational framework.
  - *Advantages*: every standard NN can be turned into a Bayesian NN. Very easy to train and no inference network necessary
  - *Drawbacks*: expensive, have to rerun model several times on data. Not very accurate (depends on activation function etc.)
- **Deep Gaussian Process**: predict mean and variance for every data point.
  - The predictive distribution is $p(y|x,X,Y) = \int p(y|x,w)p(w|X,Y)dw$
  - The likelihood term is a Gaussian $p(y|x,w) = \mathcal{N}(y; \hat{y}(x,w), \tau^{-1}I_D)$ where $\hat{y}(x,w)$ is a NN and τ^{-1} the model precision that can be derived from MC dropout
  - For the posterior, we use variational approximation: $p(w|X,Y) \approx q(w)$. In case of MC dropout, we have $\tilde{W}_i = W_i \cdot \text{diag}\left([z_{i,j}]_1^{K_i}\right), z_{i,j} \sim \text{Bernoulli}(p_i)$ where $\tilde{W}_i$ are the weights with applied dropout
  - Minimize loss $\mathcal{L} = -\int q(w) \log p(Y|X,w)dw + KL(q(w)||p(w|X,Y))$. First term is approximated by Monte-Carlo integration (equivalent to sampling dropout), and second can be approximated analytically
- Over-paramterized models give better uncertainty estimates as they capture bigger class of models. However, they also need higher dropout rates

7.2 Aleatoric uncertainty

- *Aleatoric uncertainty*: data uncertainty
- Uncertainty due to the nature of data (noise/hard to predict accurate. Example: depth estimation with bad sensor)
- Can be reduced by better data (better sensors, multiple different sensors, etc.)
- *Data-dependent/heteroscedastic aleatoric uncertainty*: specific raw inputs like images that are hard to interpret
  - Can be modeled by predicting a variance term per data point to reduce loss

$$\mathcal{L} = \frac{\|y_i - \hat{y}_i\|^2}{2\sigma_i^2} + \log \sigma_i$$

    If variance low, the loss is weighted higher, but the $\log$ term is smaller $\Rightarrow$ trade-off
- *Task-dependent/homoscedastic aleatoric uncertainty*: introduced by task like semantic segmentation or depth estimation (hard at edges). Possible solution: train on multiple tasks like edge detection
  - We can as well introduce a variance term, but shared by all data points (task individual):

$$\mathcal{L} = \frac{\|y_i - \hat{y}_i\|^2}{2\sigma^2} + \log \sigma$$

7.3 Bayes by Backprop

- Start from a NN with a distribution over its weights
- Train weights to approximate the true posterior well (similar to ELBO just with $p(\mathcal{D}) = 1 \Rightarrow \log p(\mathcal{D}) = 0$)

$$\mathrm{KL}\left(q\left(w|\theta\right)||p\left(w|\mathcal{D}\right)\right) = \mathrm{KL}\left(q\left(w|\theta\right)||p\left(w\right)\right) - \int q(w|\theta) \log p(\mathcal{D}|w) dw$$

  First term pushes distributions towards prior, and second towards modeling the data well
- Compute by Monte-Carlo integration (over distribution $q(w|\theta)$) for *both* terms:

$$\mathcal{L} = \log q(w_s|\theta) - \log p(w_s) - \log p(\mathcal{D}|w_s) \quad \text{where} \quad w_s \sim q(w_s|\theta)$$

- Example: assume a Gaussian variational posterior on the weights $w = \mu + \epsilon \cdot \log(1 + \exp \rho))$ (standard deviation with softplus trick for always positive values). Learn parameters μ and ρ per weight

1. Sample $\varepsilon \sim N(0,1)$
2. Set $w = \mu + \varepsilon \cdot \log(1 + \exp(\rho))$
3. Set $\theta = \{\mu, \rho\}$
4. Let $\mathcal{L}(w, \theta) = \log q(w|\theta) - \log p(w) p(x|w)$
5. Calculate gradients

$$\nabla_\mu = \frac{\partial \mathcal{L}}{\partial w}\frac{\partial w}{\partial \mu} + \frac{\partial \mathcal{L}}{\partial \mu}$$
$$\nabla_\rho = \frac{\partial \mathcal{L}}{\partial w}\frac{\varepsilon}{1 + \exp(-\rho)} + \frac{\partial \mathcal{L}}{\partial \rho}$$

7. Last, update the variational parameters

$$\mu_{t+1} = \mu_t - \eta_t \nabla_\mu$$
$$\rho_{t+1} = \rho_t - \eta_t \nabla_\rho$$

- In experiments, Bayesian NNs perform similar to plain NNs with dropout

8 Deep Sequential Models

8.1 Autoregressive Models

- Generative models without latent variables, but assuming an order in the data (if there is no, create an artificial order like image from left to right, top to bottom). The likelihood is the product of conditionals:

$$p(x) = \prod_{k=1}^{D} p(x_k | x_{j<k})$$

- In contrast to RNNs, there is no/not necessarily parameter sharing, and the chain cannot be of infinite length because of that
- *Advantages*: $p(x)$ is tractable
- *Drawbacks*: training and generation is slow due to being sequential and not parallel

8.1.1 NADE

- Originally defined for binary inputs/data. Can be generalized for other spaces as well
- Every output x_d is modeled by a single layer that takes as input all previous data points, and generates based on that it's prediction:

$$p(x_d = 1|x_{<d}) = \sigma\left(V_{d,:} \cdot h_d + b_d\right), h_d = \sigma\left(W_{:,<d} \cdot x_{<d} + c\right)$$

  where $V \in \mathbb{R}^{D\times H}, W \in \mathbb{R}^{H\times D}, b \in \mathbb{R}^D, c \in \mathbb{R}^H$ (H hidden dimensionality, D input dimensions)
- Objective is minimizing log likelihood: $\mathcal{L} = -\log p(x) = -\sum_{k=1}^{D} p(x_k|x_{<k})$

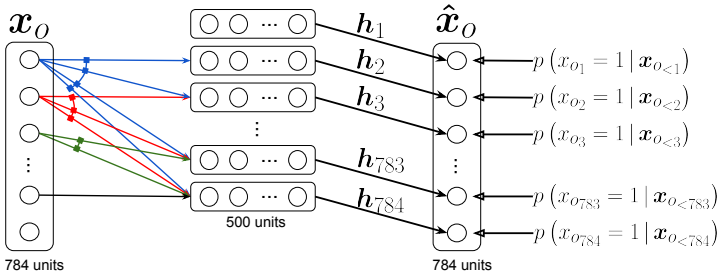


Figure 14: Concept of NADE.

- *Teacher forcing*: During training, use ground truth as input for all levels. For testing, use generated samples as input (sequentially)

8.1.2 MADE

- Use an autoencoder where we carefully mask out connections so that the output y_d only depends on inputs $x_{<d}$
- Name "autoencoder" is only because we try to reproduce the input. However, note that we neither have a bottleneck nor we try to get sparsity. We just remove connections to make the outputs depending on certain inputs

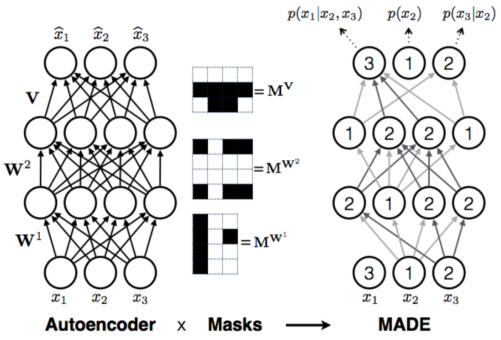


Figure 15: Masked autoencoder for autoregressive models. We set certain weights to 0 (i.e. remove connections between neurons) so that the generation of x_1 only depends on x_2 and x_3, but not on x_1 itself (which would be cheating and prevent the model of being generative).

8.1.3 PixelRNN

- Assume row-wise pixel and sequential color generation (first red channel, then green, afterwards blue):

$$p(x_i|x_{<i}) = p(x_{i,R}|x_{<i}) \cdot p(x_{i,G}|x_{i,R}, x_{<i}) \cdot p(x_{i,B}|x_{i,R}, x_{i,G}, x_{<i})$$

- Different ways of modeling it. LSTM variants mostly have 12 layers
  - *Row LSTM*: to compute next output (i.e. next hidden state), we take into consideration the three hidden states of the row above a certain pixel as "last hidden state". We get therefore a tri-angular shape of context. However, it thereby misses context from the row itself, and further away context. As it does not use pixels in the same row, the computation can be parallelized for a row.
  - *Diagonal Bi-LSTM*: Uses all pixels that were generated before by using a Bi-LSTM.

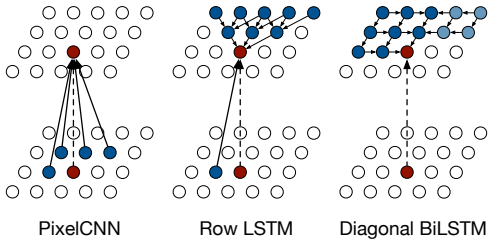


Figure 16: Comparing different methods of PixelRNN and PixelCNN. The lower level is the previous layer, and the top is the next layer. If we have a single layer PixelRNN/CNN, the lower one would be the input and the upper the generated output.

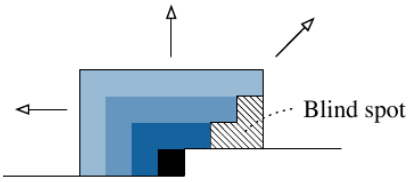
- The architecture includes residual connections to speed up training
- *Benefits*: good modeling of $p(x)$, reasonable image quality
- *Disadvantages*: slow training and slow generation

8.1.4 PixelCNN

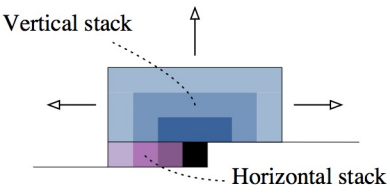
- Replace recurrence by convolutions to speed up (at least) training
- Convolutions are masked so that only context from before (i.e. left and top) can be used. See Figure 16 left and Figure 17 for an example

1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

(a) Example mask for 5×5 convolution



(b) Blindspot of PixelCNN



(c) Solution to blindspot

Figure 17: Masked convolutions in PixelCNN

- Problem: worse results than PixelRNN because of limited context and blind spot (cascaded convolutions ignore right upper part)
- Solution: use two convolutions, one vertical stack looking purely on the top part, and the horizontal stack looking to the right. Additionally, use gated convolutions (one half of the features go through tanh, the other through sigmoid)
- **PixelCNN++**: replace softmax with logistic mixture likelihood over 8 bits, use encoder-decoder architecture with skip connections

8.1.5 PixelVAE

- Standard VAE with PixelCNN as decoder/generator
- However, generator is very powerful which can lead to the problem that it ignores the latent code, and just generates "nice" images

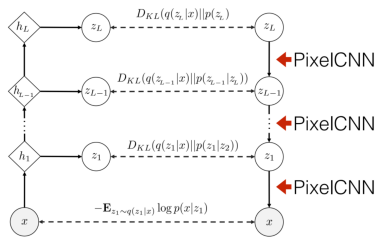


Figure 18: Architecture of a PixelVAE

9 Deep Reinforcement Learning

9.1 Fundamentals of Reinforcement Learning

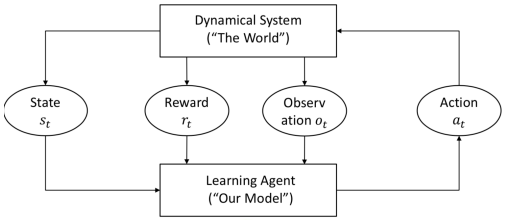


Figure 19: Interaction model between environment and agent

- The **state** s_t is the summary of all experience so far: $s_t = f(o_1, r_1, a_1, o_2, r_2, a_2, ..., o_t, r_t)$ (o_i observable part of environment at time step i). If we have a fully observable environment, then $s_t = f(o_t)$.
- The **policy** of an agent determines its actions: $\pi(a_t|s_t)$. Can be deterministic or stochastic
- The **value function** is the expected total reward under policy π:

$$q^\pi(s_t, a_t) = \mathbb{E}_\pi \left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... | s_t, a_t \right]$$

  γ as discount factor as we are most certain about close rewards and sometimes are more interested in immediate rewards

- **Bellman equation** for value function:

$$q^\pi(s_t, a_t) = \mathbb{E}_{s',a'} \left[r + \gamma q^\pi(s', a') | s_t, a_t \right] = \sum_{s'} p(s'|s_t, a_t) \cdot \left[r(s', a_t, s_t) + \gamma \sum_{a'} \pi(a'|s') \cdot q^\pi(s', a') \right]$$

- The optimal value function is therefore $q^*(s_t, a_t) = \max_\pi q^\pi(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}}$
- The **environment** can be modeled by the agent (learned from experience), and used for planning and look ahead. This can be for example a simulator

9.2 Deep RL approaches

9.2.1 Value-based approaches

- Try to learn value function q^* to get the optimal policy π^*
- The input to such models is usually the state, which should be as raw as possible (e.g. image frames). We can either add the action to the input and let the network predict its Q-value, or predict Q-values for all possible actions (second is faster and simpler)

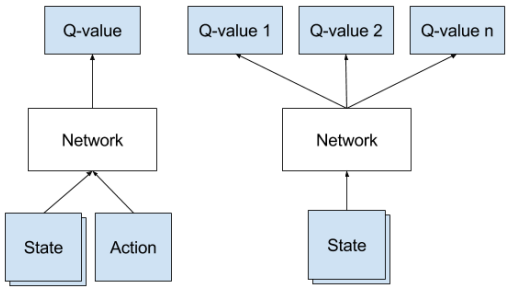


Figure 20: Modeling of Q-value predictions

- Optimization by SARSA-like loss:

$$\mathcal{L} = \mathbb{E}\left[\left(r + \gamma \max_{a_{t+1}} q(s_{t+1}, a_{t+1}, \theta) - q(s_t, a_t, \theta)\right)^2\right]$$

- For the gradients, we assume that the bootstrapped max value is fixed:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \mathbb{E}\left[-2 \cdot \left(r + \gamma \max_{a_{t+1}} q(s_{t+1}, a_{t+1}, \theta) - q(s_t, a_t, \theta)\right) \cdot \frac{\partial q(s_t, a_t, \theta)}{\partial \theta}\right]$$

- Optimize with SGD by sampling one action and state, calculate q-values for all possible future actions, and use the maximum as bootstrap goal

9.2.2 Stability problems

- As we bootstrap, the target is always changing $\Rightarrow$ policy changes fast, can lead to oscillations
- The sequential data breaks the iid assumption on which SGD relies
- The scale of Q-values is not easy to control, and is very task dependent $\Rightarrow$ gradients are unstable and can be either too large or too small
- **Improving stability**
  - *Experience replay*: store memories of $\langle s, a, r, s' \rangle$ (with e.g. a ϵ-greedy policy) in a dataset, and sample batches from there to train on. Breaks temporal dependency and helps SGD by i.i.d.
  - *Freezing target*: instead of having a moving target, we freeze the Q network every K iterations, and use that to generate our targets (Q-targets come now from a bit older network parameter setting, but is steady over K iterations). Avoids oscillations
  - *Clipping rewards*: Normalize or clip rewards to be in range $[-1, +1]$ or any other stable range. Prevents unknown scales of Q
  - *Skipping frames*: a light version of experience replay is skipping N frames between two data points to avoid too strong temporal dependency (two consecutive frames are very similar)
  - *Exploration vs Exploitation*: use a ϵ-greedy policy with annealing temperature. In the beginning, we will focus on exploration while slowly converging to exploitation

9.2.3 Policy-based approaches

- Try to learn the optimal policy π^* directly from experience (parameterized policy $\pi_w(a_t|s_t)$)
- Avoids learning the q values which are hard for continuous action spaces, and tend to oscillate because of bootstrapping
- Training steps
  1. Determine Q-value for current policy by running a simulation:
     $q^{\pi_w}(s_t, a_t) = \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... | \pi_w\right]$
  2. Maximize q-values as loss function.
     (a) If policy is deterministic:

$$\frac{\partial \mathcal{L}}{\partial w} = \mathbb{E}\left[\frac{\partial q^\pi(s, a)}{\partial a} \frac{\partial a}{\partial w}\right]$$

(b) If policy is stochastic:

$$\frac{\partial \mathcal{L}}{\partial w} = \mathbb{E}\left[\frac{\partial \log \pi^w(a|s)}{\partial w} q^\pi(s,a)\right]$$

- Asynchronous Advantage Actor-Critic
  - Learn both policy and value function
  - Multiple agents that simultaneously interact with (copy of) environment and learn
  - *Advantage estimates*: Use the learned value function to compare to your actually gained q value. Loss is therefore higher if unexpected things happen $\Rightarrow$ exploration

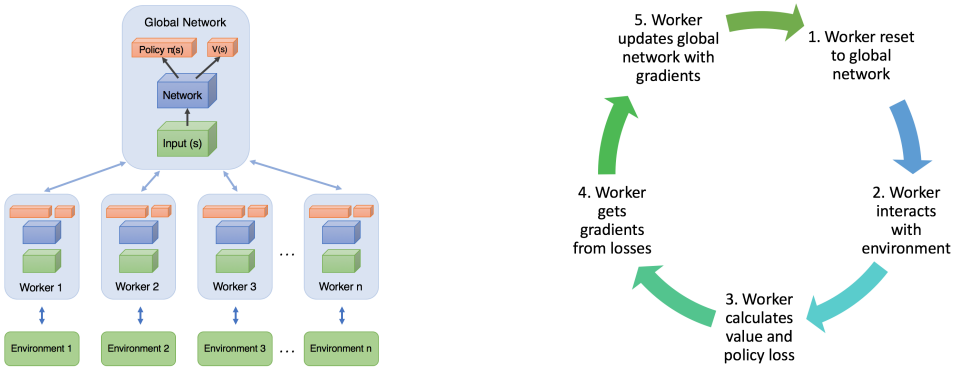


Figure 21: Schematic overview of A3C

9.2.4 Model-based approaches

- Try to model the environment to be aware of rules etc.
- Example: AlphaGo relies on Tree-Search guided by CNNs. We use two policy networks to play against each other, and one value network that predicts the value function of a state

A mostly complete chart of

Neural Networks

 asimovinstitute.org

- Input Cell
- Backfed Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Capsule Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Gated Memory Cell
- Kernel
- Convolution or Pool

Perceptron (P)

Feed Forward (FF)

Radial Basis Network (RBF)

Deep Feed Forward (DFF)

Recurrent Neural Network (RNN)

Long / Short Term Memory (LSTM)

Gated Recurrent Unit (GRU)

Auto Encoder (AE)

Variational AE (VAE)

Denoising AE (DAE)

Sparse AE (SAE)

Markov Chain (MC)

Hopfield Network (HN)

Boltzmann Machine (BM)

Restricted BM (RBM)

Deep Belief Network (DBN)

Deep Convolutional Network (DCN)

Deconvolutional Network (DN)

Deep Convolutional Inverse Graphics Network (DCIGN)

Generative Adversarial Network (GAN)

Liquid State Machine (LSM)

Extreme Learning Machine (ELM)

Echo State Network (ESN)

Deep Residual Network (DRN)

Differentiable Neural Computer (DNC)

Neural Turing Machine (NTM)

Capsule Network (CN)

Kohonen Network (KN)

Attention Network (AN)