1 Deep Neural Networks for Image Classification

The classical machine learning model consists of training a network using **training dataset** for known prediction labels, validating the model throughout training with a **validation dataset** to fine tune *hyper-parameters* and *architecture*, before finally using the trained classifier to predict classes from data with known labels from a **test dataset**, used to estimate the *error rate* of the system.

1.1 Classification

The goal of classification is to correctly predict which class $y \in \mathcal{Y}$ an input $x \in \mathcal{X}$ belongs to. In binary classification: $\mathcal{Y} = \{0, 1\}$ or $\{-1, 1\}$. In C-class classification: $\mathcal{Y} = \{0, 1, \ldots, C-1\}$.

1.2 Fully Connected Network (FCN)

For image classification, a 2D image is unwrapped to a 1D vector working as the input to a fully connected network. However, a FCN will have too many parameters, and will not be able to scale to normal size images for generalization due to not being invariant.

1.3 Generalization

High generalization of a neural network model implies that with learnt parameters on training data, the loss on newly drawn test data will be low on average. Denoting the test datasets as T_n, and trained classifier f, this implies that

$$L(f, T_n) = \frac{1}{n} \sum_{(x_i, y_i) \in T_n} \ell(f(x_i), y_i)) \longrightarrow 0 \tag{1}$$

1.4 General limitations

Some general limitations of deep learning include

- Small sample sizes ⇒ Overfitting
- Complex tasks ⇒ Memory-intensive
- Networks may be fooled
- Training data dominates learned predictor

1.5 General linear mapping for classification

Given weights W, input x, and bias b, a linear mapping for classifying the input data can be:

$$f(x) = W \cdot x + b \tag{2}$$

followed by applying an activation function.

1.6 Activation functions

The following are (commonly) used activation functions. Should be differentiable since they are differentiated in gradient-based optimization. Used to prevent / reduce weak signals to pass through network. Commonly non-linear.

1.6.1 Identity

Definition of the **identity function**:

$$a(z) = z, \quad a'(z) = 1 \tag{3}$$

Usually not used, as backpropagation has no relation to the input. Also, all layers collapse into one since the identity function is linear.

1.6.2 Logistic Sigmoid Function

Definition of the **Sigmoid function**:

$$a(z) = \sigma(z) = \frac{1}{1 + e^{-z}}, \quad \sigma'(z) = \sigma(z)(1 - \sigma(z)) \tag{4}$$

For 2-class classification problems, the output $\sigma(z) \in [0, 1]$ is the predicted probability that sample x has class label $y = 1$. Suffers from gradient saturation – its derivative gets exponentially small for large z, leading to vanishing gradients. Due to this, it usually has slow convergence and is computationally expensive. Classification using Sigmoid is called logistic regression.

1.6.3 ReLu

Definition of **ReLu** (Rectified Linear Unit)

$$a(z) = \max(0, z), \quad a'(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases} \tag{5}$$

Computationally efficient, and gradient does not saturate, but is not differentiable in $z = 0$ although not a problem in practice. Gradient is 0 for $z < 0$, causing inactive nodes to remain inactive, thus the network cannot perform backpropagation and cannot learn. Popular choice in CNNs.

1.6.4 Leaky ReLu

Definition of **Leaky ReLu**

$$a(z) = \max(0.1z, z), \quad a'(z) = \begin{cases} 0.1 & z < 0 \\ 1 & z \geq 0 \end{cases} \tag{6}$$

Unlike standard ReLU, it fixes the dying ReLU problem, enabling backpropagation for all values. A generalization of Leaky ReLu is called the Parametric ReLu, using any $\alpha \in \mathbb{R}$ instead of the constant value 0.1.

1.6.5 Hyperbolic tangent

Definition of **hyperbolic tangent** function:

$$a(z) = \tanh(z), \quad a'(z) = 1 - a(z)^2 \tag{7}$$

The output is $a(z) \in [-1, 1]$, i.e., zero centered making it easier to model inputs with strongly negative, neutral and strongly positive values. However, suffers from vanishing gradients and slow convergence for extreme (positive and negative) values of gradients.

1.6.6 Sinusoid

Definition of the **Sinusoid** function:

$$a(z) = \sin(z), \quad a'(z) = \cos(z) \tag{8}$$

Centered around zero. Because Sine is a periodic function, low and high input values might produce same output. Is not monotonic, which is a essential feature of activation functions in image classification according to the universal approximation theorem.

1.7 General Formulation of Forward Pass

Given weights w, activations a, bias b, and activation function g, the general formula for the forward pass is:

$$a_k^{[l+1]} = g\left(\sum_{j=1}^{n^{[l]}} w_{jk}^{[l+1]} a_j^{[l]} + b_k^{[l]}\right), \qquad k \in \{1, 2, \ldots, n^{[l+1]}\}, \quad l \in \{1, 2, \ldots, L\} \tag{9}$$

1.7.1 Softmax

At the **last layer** before the output, the activation function will in general not provide outputs which represent probabilities.

A solution is to apply the **Softmax** function, defined as:

$$a(z)_k = \frac{e^{z_k}}{\sum_{i=1}^{n} e^{x_i}} \tag{10}$$

Then, the activations will represent a probability, because $\sum_k a(z)_k = 1$.

If Softmax is causing numerical instability two common tricks that help are:

- Shifting exponential arguments based on maximum z, $\max(z)$.
- Taking logarithm of exponential $\Rightarrow$ Gets rid of division.

2 Training a Neural Network

2.1 Loss functions

2.1.1 Cross Entropy / Log loss

Given training samples x_i, predicted as $\hat{y} = \sigma(f(x))$ and associated class labels y_i, $i = \{1, 2, \ldots, n\}$, it is common to apply the cross entropy loss function. The loss function is derived as the function to optimize by maximum likelihood estimation.

The **cross entropy loss**, also known as the **logistic loss** is defined as

$$\ell(y, \hat{y}) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i). \tag{11}$$

For one-hot labels where $y_i \in \{0, 1\}$, the cross-entropy loss reduces to the more simple form:

$$\ell(y, \hat{y}) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) \quad \text{if } y_i = 1 \tag{12}$$

or

$$\ell(y, \hat{y}) = -\sum_{i=1}^{n} (1 - y_i) \log(1 - \hat{y}_i) \quad \text{if } y_i = 0 \tag{13}$$

2.1.2 L_1 loss

For image classification or in general, the L_1-loss is defined as

$$\ell(y, \hat{y}) = \sum_{i=1}^{n} |y - \hat{y}| \tag{14}$$

2.1.3 L_2 loss

For image classification or in general, the L_2-loss / quadratic loss / regression loss is defined as

$$\ell(y, \hat{y}) = \sum_{i=1}^{n} (y - \hat{y})^2 \tag{15}$$

2.1.4 Hinge-loss

For image classification, the hinge-loss is defined as

$$\ell(y, \hat{y}) = \max(0, 1 - y\hat{y}), \quad y \in \{-1, 1\} \tag{16}$$

Low hinge loss $\Rightarrow$ low zero-one loss, and it is fine to minimize average hinge loss.

2.1.5 Focal loss

See Object Detection.

2.2 Gradient descent

The gradient decent algorithm proposes to solve the minimization problem

$$(w^*, b^*) = \arg\min_{w,b} L(w, b) = \arg\min_{w,b} \frac{1}{n} \sum_{(x_i, y_i)} \ell(f(x_i), y_i), \tag{17}$$

where ℓ is a loss function, to find the optimal parameters w^* and b^*.

The **gradient descent algorithm** is as follows:

Given step size / learning rate η, and initial vector w_0

while change in function $\Delta g(w) = \delta_g$ ¿ δ_{stop}:

$w_{t+1} = w_t - \eta \nabla_w g(w_t)$

Compute $\delta_g = \|g(w_{t+1}) - g(w_t)\|$,

where $g(w)$ is some function, in this case the loss function L.

Converges when $\nabla_w g(w_t) \approx 0$, i.e. local or global minimia, or a saddle point. Problems include choice of learning rate η. Too large $\eta \Rightarrow$ divergence and no solution, while too small $\eta \Rightarrow$ slow convergence and computationally slow. Using the full training data sample is also called **batch gradient descent**.

2.3 Stochastic gradient descent

Instead of batch gradient descent, **stochastic gradient descent** (SGD) computes the gradient only over a *randomly* selected subset of samples; over **mini-batches**.

SGD when starting at index m using k samples will compute the gradient of

$$\nabla_w \left(\frac{1}{k} \sum_{i=m}^{m+k-1} \ell(f_w(x_i), y_i) \right) \tag{18}$$

Improvement from batch GD, which is too costly to compute all gradients. SGD is a noisy approximation of GD, and the noise helps in preventing overfitting – works as a sort of regularization, preventing the model to study specific data too closely. SGD performs better than GD finding good local optima, due to higher variance in input data.

2.4 Back-propagation

In backpropagation, gradients are computed throughout the network via the chain rule, along the edges in the network graph, starting at the output(s) $\hat{y}_j$ and propagating all the way back to the input data x_i.

3 Optimizing a Neural Network

By optimizing the neural network, we introduce methods that improve the way the parameters are updated when applying gradients.

3.1 Learning rate (η)

3.1.1 Learning rate scheduler

The learning rate η can be chosen as constant. However, in a flat region steps can be very small, implying slow convergence. In contrast, in steep regions, the step may become too large, and the model may overshoot. One solution to decrease the learning rate in flat regions is to use **learning rate decay**.

Alternatives are **step-wise** and **polynomial** learning rate decay:

$$\textbf{Step-wise:} \quad \eta_{t+1} = \begin{cases} \eta \cdot \gamma, 0 < \gamma < 1 & \text{if } t = c \cdot K, c \in \mathbb{N} \\ \eta_t & \text{else} \end{cases} \tag{19}$$

$$\textbf{Polynomial:} \quad \eta_t = \frac{\eta_0}{t^\alpha}, \alpha > 0 \tag{20}$$

where K determines the frequency (after K steps).

3.1.2 Learning rate warm-up

Similar to decay, but resolves instabilities at the start of learning phase by starting slower during the first K epochs, before switching to learning rate decay as described above.

3.2 Weight decay

By weight decay, we add a regularization term on the weights, such that the update during gradient descent becomes

$$w_{t+1} = w_t(1 - \lambda\eta_t) - \eta_t \nabla_w L(w_t) \tag{21}$$

This shrinks the weight towards zero. For SGD, this is the same as Ridge/L_2 regularisation.

3.3 Momentum and Nesterov Momentum

The idea behind momentum has a physical interpretation. It acts as a memory for gradients in the past, by remembering larger step-sizes from past steps in flat regions. It can be interpreted as a rescaled, time-dependent weighted average of gradient, which directs the gradient towards the local minima. Introduces new hyper-parameter α, often set to $\alpha = 0.9$.

The **standard momentum update** is the following update:

$$m_0 = 0, \alpha \in (0, 1)$$

$$m_{t+1} = \alpha m_t + \eta_t \nabla_w L(w_t)$$

$$w_{t+1} = w_t - m_{t+1}$$

An extension of this is the Nesterov momentum update, which computes the gradient at the approximate next position: $w_t - \alpha m_t$.

The **Nesterov momentum update** is the following update:

$$m_0 = 0, \alpha \in (0, 1)$$

$$m_{t+1} = \alpha m_t + \eta_t \nabla_w L(w_t - \alpha m_t)$$

$$w_{t+1} = w_t - m_{t+1}$$

Both methods are used to avoid small local minima and saddle points, to not overshoot the global or a good local minima.

3.4 RMSProp

RMSProp is based on the **Exponential moving average** (EMA), which generalizes the idea behind momentum by considering a time series of past gradients. EMA utilizes gradients in such a way that gradients **far away have less effect**, with the same intuition as momentum. RMSProp deals with flat regions, i.e., where the gradient $g_t = \nabla_w L(w_t)$ is very small. One idea is divide the gradient by it's norm to normalize the gradient. However, we can used EMA, which considers past gradients and instead divide by that.

RMSProp proposes the following update:

$$w_{t+1} = w_t - \eta_t \frac{g_t}{\sqrt{\text{EMA}(\|g_s\|^2)_t + \varepsilon}}, \tag{22}$$

where g_s is a time series of gradients. The ε assures numerical stability in case gradient are near-zero.

RMSProp introduces two additional hyper-parameters, α inside EMA$(\cdot)$ and ε. RMSProp has the following features:

- Divides gradient by history of gradients, over a time-limited horizon
- Upscales step size in flat regions

- Downscales step size in steep regions

3.5 ADAM

ADAM (Adaptive Momentum Estimation) is a combination of RMSProp with a momentum term, but with additional ideas making it an **improvement over RMSProp**:

- Normalize every dimension of the update
- Turn the EMA terms into true weighted averages

The first idea is achieved by squaring every single dimension. The second idea is achieved by multiplying the EMA terms with a appropriate normalizer.

An alternative to ADAM is **AdamW** – Adam with decoupled weight decay. The additional weight decay term is added to the parameter/weight update:

$$-\lambda \eta_t w_t \tag{23}$$

The improvement from Adam is that AdamW performs **stronger weight decay** for large gradients.

3.6 Gradient clipping

Standard gradient clipping is to keep gradient norms bounded:

$$Clip_\lambda(g) = \begin{cases} g & \text{if } \|g\| \leq \lambda \\ \lambda \frac{g}{\|g\|} & \text{if } \|g\| > \lambda \end{cases} \tag{24}$$

This will bound the gradient norm by $\|g\| \leq \lambda$ An alternative is **adaptive gradient clipping**, which also considers the weight vector norm, $\|W\|$.

4 Hyperparameters

- η – Learning rate
- N_{epocs} – Number of epochs
- $N_{\mathcal{B}}$ – Batch size
- λ – Ridge and Lasso regularisation penalty term
- S – Stride
- P – Padding
- D – Dilation (growth) factor

- F - Filter size ($F \times F$)
- p – Dropout probability
- α - Momentum parameter
- ε - Stabilizer in RMSProp

5 Initialisation of a Neural Network

5.1 Input initialisation

Images are usually represented by sub-pixels $\in [0, 255]$. However, the neural network expects inputs $\in [0, 1]$. Thus, a common pre-processing step is to perform **input normalization** using the training set mean and standard deviation ($\mu_{train}, \sigma_{train}$):

$$subpixel[c] = \frac{subpixel[c] - \mu_{train}[c]}{\sigma_{train}[c]}, \tag{25}$$

where c is the channel. Another pre-processing step may be **resizing the image** so it ends up with the desired input aspect ration that a pre-trained network might be specific to. Alternatively, a **random crop** or **center crop** can be performed to satisfy the size requirements. To classify over a larger input than normal, the network can be modified to accept this input size. This will **change the number of parameters** in fully connected layers, but **not change the number of parameters** in convolution layers. It is here possible to use **global / adaptive pooling**, which are methods where the output size is fixed, and the pooling kernel size changes depending on the input to match desired output size.

5.2 Data augmentation

The idea of data augmentation is to extend the training data available during **training**, and to perform model evaluation over averaged augmentations of the same input during **testing**. Examples of data augmentation are:

- Random, center and corner crops
- Vertical flips
- Shearing
- Translation
- Rotations
- Distortions of brightness, contrast, gamma

- Adjusting hues
- Adding noise (Gaussian, white)
- Adjusting speed (Audio)
- Adjusting frequency (Audio)

Augmentation can also be applied to **ground truth data**, e.g. adding noise to label, moving boundary boxes, or moving segmentation boundaries. Data augmentation provides plausibly derived data, and are used improve the models invariance to different features.

5.3 Weight and bias initialisation

Weights are chosen with the following σ such that:

- $\sigma^2_{feature\,map}$ are similar across layers (forward pass)
- $\sigma^2_{gradient\,norm}$ are similar across layers (backward pass)

Random weights are selected to avoid ending up with same forward pass values ⇒ Same gradient update ⇒ Neurons learn to encode same feature.

5.3.1 Xavier initialization

For **tanh** networks, a common way to initialize the network is to set bias $b = 0$, and initialize weights with standard deviation

$$\sigma = \sqrt{\frac{1}{n}}, \qquad \text{for } n \text{ samples.} \tag{26}$$

Helps avoid slow convergence, and avoids exploding and vanishing gradients by not setting the weights close to 1.

5.3.2 He initialisation

For **ReLu** networks, a common way to initialize the network is to set bias $b = 0$, and initialize weights as random values for symmetry breaking, also depending on the number of input samples. Draw weights from a normal distribution with standard deviation:

$$\sigma = \sqrt{\frac{2}{n}}, \qquad \text{for } n \text{ samples.} \tag{27}$$

5.4 Weight Standardisation

Standardizing the weights is motivated by BatchNorm decoupling the gradient flow from scale of weight. Has in combination with Group normalization and a small mini-batch size of 1 outperformed BatchNorm.

6 Overfitting

Overfitting is when the loss on new unseen data points, such as the validation and test data, is much higher than the loss on the training data. With enough parameters, you can always approximate your training data sufficiently good due to the universal approximation theorem, but this might still overfit the validation and testing dataset. This is because having the ability approximating any function, does not mean we can learn any function well from finite training data (overfitting). Hence, validation and testing datasets need to be disjoint, also to improve generalization. Overfitting breaks the central limiting theorem, because in that case the model is specific to a particular training set.

6.1 Lasso Regularisation (L_1)

A proposed regularisation of a neural network is the **Lasso regularization**. The following term is added to the total loss:

$$\lambda \sum_{j=1}^{m} |\theta_j|, \tag{28}$$

where λ is a regularization penalty. This corresponding to the L_1-norm of the weights and/or biases represented by θ. Because of the non-differentiability of the absolute value at zero, it can potentially set the parameters $\theta = 0$.

6.2 Ridge Regularisation (L_2)

There are several ways to penalize a network. One way is through **Ridge regularization**, which penalizes the weights by avoiding them getting too large. The following term is added to the total loss:

$$\lambda \sum_{j=1}^{m} \theta_j^2, \tag{29}$$

Ridge regularisation penalizes very large coefficients to better generalize on new data, i.e. suppresses overfitting. In contrast to Lasso regularization, the parameters θ cannot be set to zero, only decreased in size. Bias is increased for lower variance.

6.3 Dropout

A dropout layer is a way of adding noise to a learning problem. Consider dropout probability p. Then, during **training** (1-p) of the neurons within a layer are set to zero. During **testing**, it works as a rescaled identity. Prevents the model from using non-generalizing correlations between neurons – this reduces the statistical correlation between features, and the model is not biased for one single correlation.

6.4 Early Stopping

Early stopping is a form of regularisation used to **avoid overfitting**. This can be achieved by either:

- Training the model for a finite / preset number of epochs (Naive)
- Stopping when the loss function update is small, $\delta_w \approx 0$.
- Validation set strategy – stop the model when the validation error starts increasing, indicating that the model has started to overfit.

Although the validation set strategy is the best one in terms of preventing overfitting, it might be the most computational expensive one. Alternatively, a hybrid between the strategies may be proposed.

6.5 Other Methods to Prevent Overfitting

Other methods that minimize overfitting include:

- SGD – Noise from SGD works as regularisation
- Weight decay
- Data augmentation
- Cross-validation
- Removing FC layers
- Training with more data
- Ensembles models – combining predictions from multiple separate models

7 Convolutional Neural Networks (CNNs)

In convolution, a kernel (matrix) is used for blurring, sharpening, detecting edges, and more by convolution between a kernel ($N_F \times N_F$) on a image ($N_H \times N_W$). The filter coefficients (weights) will be learned during training, including potential biases.

7.1 FCN for CNN?

Consider input x to be each pixel value of the image, unwrapped from 2D to 1D. For an RGB image of size 512×512 this would result in

$$512 \times 512 \times 3 = 786,432 \text{ input nodes}, \tag{30}$$

resulting in millions of parameters! Also, a FCN does not inherit translational invariance, and does not utilize local analysis, which is sufficient due to background clutter.

7.2 Stride

Stride, S, is the spatial step length in convolution. Affects the size of the activation map / feature map. Downsamples with a factor equal to the stride. Adding stride is often combined with increase in number of channels, C

7.3 Padding

Padding, P, adds ghost cells / pixels outside the image which the kernel operates on. Most common approach is to **zero pad**. Alternatives include: value of nearest pixel, mean pixel value, mirror-reflecting indexing, circular indexing.

7.4 Dilated convolutions

Dilation factor of 2 implies putting a 0 between every pair of elements. This increased the receptive field, but does not reduce the spatial size of the activation map. Mimics larger filters, without increasing number of parameters. You can also chose a growing dilation factor, to get even larger receptive fields, similar to increasing the stride ($S > 1$).

7.5 Pooling

Operation over the activation / feature map, over all channels. Results in downsampled image and provides local translational and rotational invariance. Can also be applied to the total image.

Max pooling: Select maximum value within filter neighborhood. Some benefits of using max pooling are:

- It reduces the spatial size of the input, thus reducing memory constraint
- Builds scale invariance
- Builds local rotational invariance
- Has no additional trainable parameters
- Adds a non-linear operation (a standard convolutional layer is linear)
- Extracts pronounced features (e.g. edges)

However, it may **remove important information** in the forward pass, and **gradients** may become **zero** leading to slower training.

Average pooling: Average over all values within filter neighborhood. Advantages of average pooling are:

- Builds some translational invariance
- Extracts features more smoothly than max pooling

Global average pooling: Average over all values within a feature map. This is usually performed **instead of adding a fully connected layer** or **flattening** at the end of a CNN, performed over all feature maps, resulting in a vector, which is fed to the Softmax layer. Advantages of using global average pooling are:

- No additional parameters to optimize
- Overfitting is avoided
- Keeps translational invariance of the CNN

7.6 Receptive Field / Field of View

The receptive field of a CNN layer is how much of the input that a particular neuron is influenced by. Multiple small filters give the same receptive field as one larger. However, many small filters will lead to larger memory footprint during training.

Stride has an effect on the **theoretical receptive field**, and can be computed as follows:

$$R^{[k]} = R^{[k+1]} + (F^{[k]} - 1) \prod_{i=1}^{k-1} S^{[i]}, \qquad R^{[0]} = 1, \tag{31}$$

where R is the receptive field, F is the filter size, S stride.

Because the pixels at the center of a receptive field have a larger impact on the output, they propagate information to multiple neurons, compared to boundary pixels. Thus, during backpropagation, center pixels have much larger gradient magnitude from that output. Consequently, the **effective receptive field** is much smaller than the full theoretical receptive field. This can also be affected by using a ReLu activation function which can produce zero output, essentially killing the neuron.

7.7 Feature Map Size / Output Size

Output sizes can vary between the following:

- **Valid**: Output image will be reduced in size. Only positions where filter fits in image is included.

- **Same**: Input image size = Output image size
- **Full**: Output image will be increased in size Filter will have complete overlap with the image.

For deep learning, **Valid** or **Same** is commonly used. For **Same** and **Full**, padding is needed.

We can calculate the spatial size of the feature/activation map as follows:

$$N^{[i+1]} = \left\lfloor \frac{N^{[i]} - F + 2P}{S} + 1 \right\rfloor, \tag{32}$$

where P is the padding, S the stride, F the spatial size of the filter, and N is the spatial size of the feature/activation map.

7.8 Final CNN layer

In the final layer, two approaches are common:

- Want C outputs, then apply $N_C = C$ filters in the last layer, and average over spatial dimensions, and apply Softmax. (Global average pooling)
- Flatten the 3D matrix, stack FCN(s) where the last has C nodes, and apply Softmax.

8 Improving CNNs

CNNs are translational invariant and more suitable for image analysis compared to FCN, but may still be insufficiently invariant to:

- Occlusion
- Varying illumination
- Clutter
- Size
- Rotations
- Deformation
- Interclass variation

8.1 Data augmentation

Through data augmentation, CNNs can "remember" specific features such as:

- Texture
- Shape
- Rotation
- Size

texture, shape rotation and size.

8.2 Choice of Kernel

The kernel or filter is applied locally to all pixels in the image, thus facilitating **translation invariance**. Also drastically reducing number of trained parameters, since the filter is reused. Filter size should also be small, as this results in less parameters. During convolution $C = N_C$ filters are applied to achieve C outputs.

8.3 Depth-wise separable convolution

A technique composed of two steps - depthwise and pointwise convolution, which is parameter efficient. **Depthwise convolution** takes an input shape $[N_H^{[i]}, N_W^{[i]}, N_C]$, applies N_C different filters, and outputs $[N_H^{[i+1]}, N_W^{[i+1]}, N_C]$. **Pointwise** convolution takes input shape $[N_H^{[i]}, N_W^{[i]}, N_C^{[i]}]$ and applies $F_N = N_C^{[i+1]}$ filters acting on the same pixel throughout all channels, resulting in shape $[N_H^{[i]}, N_W^{[i]}, N_C^{[i+1]}]$ The number of parameters can be reduced by **ten-fold**.

8.4 Batch Normalisation

Batch normalization (Batch Norm / BN) learns to output values which have constant mean μ, and constant standard deviation σ, or equivalently constant variance σ^2. By normalizing the outputs, it reduces the change of vanishing gradients, lessens inter dependencies between distributions, and makes the network learn faster and more stable. Performing BN will also make the training and test dataset more (statistically) similar, which helps **avoid overfitting** to the statistics of the training data. Works well for CNNs, but for RNNs **layer normalization** is better.

The **batch normalization** algorithms with input x_i over the mini-batch $\mathcal{B} = \{x_1, x_2, \ldots, x_m\}$ seeks to learn he parameters γ and β, with output $y_i = \mathbf{BN}_{\gamma,\beta}(x_i)$:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// Compute mean}$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}}) \qquad \text{// Compute variance}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \varepsilon}} \qquad \text{// Normalize}$$

$$y_i = \gamma\hat{x}_i + \beta = \mathbf{BN}_{\gamma,\beta}(x_i) \qquad \text{//Scale and shift}$$

This is valid during **training**.

During **training** and validation, the running mean and variance are updated, used during testing.

During **testing**, BatchNorm performs two steps:

- $\hat{x} = \frac{x - \mu_{run}}{\sqrt{\sigma^2_{run} + \varepsilon}}$ // Normalize
- $y_i = a\hat{x} + b$ //Scale and shift

where a and b are the learnt rescaled parameters, and μ_{run} and σ^2_{run} usually a moving averages of the mini-batches from training. Alternatively, they can be recomputed over the full batch, but this is computationally expensive.

For testing, it simulates that the test sample comes from a batch which has mean and variance equal to the training data.

8.5 Group normalization

Instead of computing over a large mini-batch, **group normalization** proposes to compute the statistics over a subset of filter channels in the feature map. Has show to outperform BatchNorm in combination with weight standardization and batch size = 1, which is a huge ease on the GPU memory when using a large training set.

8.6 Finetuning

Finetuning is a type of transfer learning, which uses a pre-trained neural network model. Finetuning pre-initializes the network with some pre-trained features that has been show to perform well on another task. Based on empirical observation, low-level features in a neural network learnt over wide and general tasks can be reused for other tasks. The only thing we need to change is **input layer** to support the input dimension, and/or the **last layer** to the number of output classes in the proposed problem. Hence, last

layer does not use pre-trained weights. Another option is to only train the last set / top layers, which can be an beneficial for *very small datasets*.

8.7 Teacher-student training

Teacher-student training is a type of transfer learning, where a teacher network is used to train / "teach" a student network to make the same prediction as the teacher.

8.8 Multi-task learning

Multitask learning is a type of transfer learning, where an ensemble of models (multiple models) are combined in order to predict multiple things. An example is to ensemble one model trained to classify numbers, with one model trained to classify angles, and use a multitask model to predict both numbers and angles of e.g. the MNIST dataset.

9 Residual Neural Networks (ResNets)

Networks with **residual/skip connections** are called residual neural networks. The residual connection lets gradient flows as the identity through the shortcut, preventing vanishing gradient problem:

If the residual connection is two convolution layers (C_1 and C_2) upstream, where the input data z is provided, the forward pass and its derivative during backpropagation will be as follows:

$$f(z) = z + C_1(C_2(z)) \quad \Rightarrow \quad f'(z) = 1 + \nabla C_1(C_2(z)) \tag{33}$$

Hence, if $\nabla C_1(C_2(z)) \approx 0$, we avoid vanishing gradients.

The residual connection also provides a shortcut in the forward pass, so convolutions across the parallel path can learn additional non-linear function on top. Also easy to unlearn to identity in case of poor fit – convolution weights can be set to zero and only use the identity, while relearning the weights. An extreme version of residual neural networks is **Densenets** where the input data is propagated forward to (almost) every convolutional layer within a "Dense block", and has shown to be parameter efficient among heavy networks.

10 Performance Estimation and Metrics

After a model has trained it is important to evaluate the accuracy and robustness of the trained model. However, performance estimation on the training dataset will likely provide **overoptimistic** estimates. The model may perform well on training data, but

have low performance on new data. Instead split the data into a training, testing, and also a validation subset, with a split partition of 70%, 15% and 15%, respectively.

10.1 The Training Subset

The training subset is used purely **for training models**.

10.2 The Testing Subset

The test subset is used to **evaluate the final model**. Ideally, only evaluate on the test set once, keeping it "locked away" until the single (ensemble) model is fully trained. If the model is to be tested/evaluated again, a completely new test set should be used.

10.3 The Validation Subset

The validation subset can be used for **tuning hyper-parameters** and **adjusting the architecture** of the model.

10.3.1 K-fold cross validation

Instead of a fixed validation set, it can be split into multiple subsets. The idea of **K-fold** cross-validation is to randomly partition the validation subset into K folds of equal size. Then repeat the following K times: Use the K^{th} fold for **validation**, and the rest for **training**. This results in K models, which can be used in an ensemble model, where a combination of models are used to classify and a combination of the prediction is selected. However, training and validation time will be multiplied by K, although acceptable for small K. K-fold cross-validation is a **resampling technique** for validation, using the data more efficiently than just applying a fixed validation subset. With only one validation set, the best performance might be overoptimistic.

10.4 Representability

Usually, we work on a **development dataset** which is often fairly representative of one (or more) settings where we want our model to work. Introducing an **external dataset**, which is usually not representative of the intended application can give a good indication of how well the trained model generalizes. Again, the external dataset should be only used once to evaluate the final model.

10.5 Generalisation

To improve the models generalisation and facilitate learning, there are multiple approaches we can take.

Controlling the **network's capacity**. This can be done by: (1) Stacking layers with small kernels, (2) Depthwise separable convolutions, (3) Reduce width and depth of layers, (4) Dropout, (5) Weight decay.

We can **facilitate learning** by using: (1) Residual connections, (2) BatchNorm, (3) Transfer learning (Finetuning), (4) Learning rate scheduler, (5) Optimisation method.

For image based networks, we can perform **image-specific normalisation**, thus making images more similar. E.g. compute mean and standard deviation of all values and normalize images based on this. Either for the particular image or for all values/pixels of a particular image.

For image based networks, we can perform **data augmentation**, to both increase the dataset sizes, but also introduce variations of the images which may improve the models invariance.

There is a trade-off between facilitating the learning of relations that generalize very well, which includes on external dataset, and not occluding the relevant information for the prediction task.

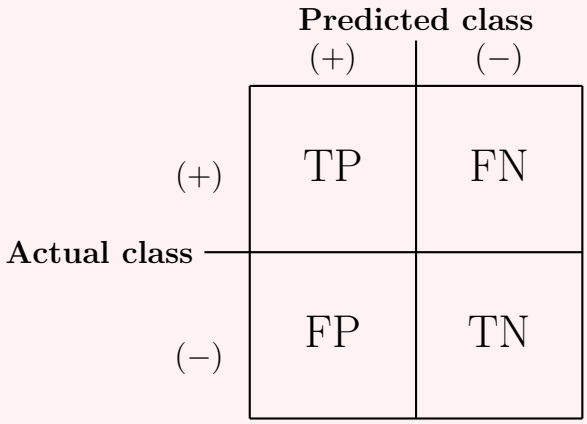
10.6 Performance metrics

Metrics should be chosen by convention, and should reflect the intended application. A classification metric such as balance accuracy may reflect the intended application. Using classifications rather than prediction scores results in **lower performance estimate**. Also, for many performance metrics it is relevant to find the uncertainty of the performance estimate, since they will always be associated with an uncertainty. This can include estimating the 95% confidence interval, where for some metrics there exists specifics procedures for estimating this.

10.6.1 Confusion Matrix

The confusion matrix consists of elements that specify the number of samples of a class A was predicted as a class K, where the diagonal elements specify the correctly classified samples.

In the case where we have **two classes**, the confusion matrix reduces to:



Where TP is true positive, FN is false negative, FP is false positive, and TN is true negative. Diagonals indicate correctly predicted classes.

10.6.2 Metrics

Here we present several metrics deduced from the confusion matrix for **two classes**.

The **precision** is defined as:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{34}$$

This is a measure of positive predictions that are actually positive.

The **recall** or **sensitivity** or true positive rate (TPR) is defined as:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{35}$$

Recall measure proportion of actual positives that are positive predictions.

The **specificity** is defined as:

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{36}$$

Specificity measure proportion of actual negatives that are negative predictions, i.e. the true negative rate (TNR). Also (1 - specificity) is known as the false positive rate (FPR).

10.7 Precision-Recall Curve

Depicts pairs of precision and recall for different thresholds. Starting with larger threshold than all scores, gradually decrease until threshold is below all scores and draw a line between the new (recall, precision) pair, forming the **precision-recall curve**.

10.8 Average Precision (AP)

The **average precision** (AP) is the area under the precision-recall curve, where higher is better. Furthermore, the **mean average precision** (mAP) is the mean of the AP in case there are *multiple classes* (> 2).

10.9 Receiver Operator Characteristic (ROC) Curve

The **ROC** curve is created from the Sensitivity (Recall) and (1-Specificity). Starting at the origin and ends at (1,1).

10.10 Area Under the Curve (AUC / AUROC)

The **AUC** or **AUROC** is the area under the ROC curve. a The AUROC can be viewed as the probability that the predicted score for the positive class is larger for an actual positive sample, than for an actual negative sample.

11 Adversarial Attacks

In adversarial attacks, the goal is to take an image and create a modified image which looks very similar, but becomes **misclassified** when running it through the classified.

11.1 Targeted attacks

A targeted attack will consider an input image x, a multi-class classifier $f(x)$, such that x is predicted for a class $c = \arg\max_c f_c(x)$. Then, the goal of a targeted attack is to create a similar image $z \approx x$, such that we predict another specific class $a \neq c$.

The **targeted attack** problem can be stated as: create image $z \approx x$ such that:

$$a = \arg\max_c f_c(z) \tag{37}$$

$$f_a(z) < \max_{c \neq a} f_c(z) \tag{38}$$

$$\|x - z\| < \epsilon \tag{39}$$

where $f_c(\cdot)$ can be either a Softmax layer or the Logits from the last layer before it, and is the prediction for class c.

11.2 Untargeted attacks

Untargeted attacks aim to misclassify an image x relative to its originally predicted label, i.e. we are not targeting any certain fixed class.

11.3 Combining targeted and untargeted attacks

To achieve misclassification we can compute the gradient and perform gradient descent (**untargeted**) on $f_a(z)$:

$$z_{t+1} = z_t - \eta \nabla f_a(z_t). \tag{40}$$

This means we minimize the score of $f_a(\cdot)$, i.e., of predicting the correct class. This can also be combined with a **targeted** approach, where we maximize the score for another class $c \neq a$, with classifier $f_c(\cdot)$ and perform gradient ascent:

$$g_t = \arg\max_{c \neq a} \nabla f_a(z_t) \tag{41}$$

$$z_{t+1} = z_t \underbrace{-\eta \nabla f_a(z_t)}_{\text{Gradient descent on } f_a(z)} \underbrace{+\eta \nabla f_{g_t}(z_t)}_{\text{Gradient ascent on } f_{g_t}(z)} \tag{42}$$

This is a white-box attack.

Note that we are updating the *image*, not any parameters. We can do this because decision boundaries are not smooth, rather they are very complex and not specified by training, and may be exploited through adversarial attacks by creating samples in this outlier region.

11.4 Black box attacks

In **black box** attacks, the attacker has access to only the outputs of prediction models.

11.5 White box attacks

In **white box** attacks, the attacker has access to all details of the prediction model, which includes gradients, layers, and training data. Usually performed by computing a gradient of some sort.

11.6 Universal adversarial attacks

In a **universal adversarial attack** the goal is to create a **universal adversarial vector**, which is a perturbation that can be applied to many input samples, and has a relative high change to cause misclassification in a large percentage of them.

11.7 Iterative Gradient Method

Iterates until misclassification is achieved, either through gradient ascent or descent. Is a *white box* approach, both *targeted or untargeted*. Uses logits for computing probabilities, since Softmax is problematic if $f_c(x) \approx 1$, then gradients are close to zero.

Targeted iterative gradient will perform gradient ascent towards a target class:

$$x_{n+1} = x_n + \varepsilon \nabla f_c(x_n) \tag{43}$$

Untargeted iterative gradient will perform either gradient descent away from the target class:

$$c^0 = \arg\max_c f_c(x_0) \tag{44}$$

$$x_{n+1} = x_n - \varepsilon \nabla f_{c^0}(x_n) \tag{45}$$

or gradient ascent towards the least probable class:

$$c^* = \arg\min_c f_c(x_0) \tag{46}$$

$$x_{n+1} = x_n + \varepsilon \nabla f_{c^*}(x_n) \tag{47}$$

11.8 Fast Gradient Sign Method

The fast gradient sign method is a *white box, untargeted* method that uses the sign of the gradient of the training loss, and uses only one iteration. Depending on the sign, it will maximise the loss between the prediction on x_n and its true label, resulting in misclassification. However, selecting ε can be tricky, and may change the image too much, to the degree where a human would notice.

The **fast gradient sign method** for a predicted class label c^* is:

$$x_{n+1} = x_n + \varepsilon \text{sign}\left(\nabla_x L(c^*, f))(x_n)\right), \tag{48}$$

where ε is chosen such that **1 iteration** is sufficient, thus being *fast*.

An alternative to the fast gradient sign method is the **iterative gradient sign**, which performs the technique over multiple iterations, by either (1) walking in the direction of least likely prediction class, or (2) by maximising the loss to its true class.

11.9 Surrogate attacks

Surrogate attacks are a type of *black box* attack, where the goal is to train an approximation $g(x)$ of the classifier $f(x)$, such that $g(x) \approx f(x)$, and attack the surrogate, hoping it generalizes to the actual target network. Iterative method.

11.10 Boundary attacks

Boundary attacks are a type of *black box* attack, where the goal is to feel your way along the decision boundary while moving closer to the sample which one wants to corrupt, while staying on the wrong side of the boundary. Does not use gradients, instead uses rejection sampling to define moves, where perturbations are sampled from a distribution. The perturbations are accepted in case it gets closer to the target.

11.11 Defences

Two examples of defences to adversarial attacks:

Defence 1: **Adversarial hardening** – makes the model more robust by training it with both normal and adversarial examples. Still, damage can be done even if 1% of attacks pass.

Defence 2: **Adversarial detection** – measures the change of prediction on input x when the input is transformed, and detect attacks by noticing large changes for adversarial perturbations compared to clean data.

12 Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) are used to handle a **time series** of input data, which standard CNNs are limited by since they cannot process data of unknown length. Examples are text, images (video frames), and time series.

12.1 Vanilla RNN

A vanilla RNN takes a time series of input data x_t and outputs y_t, as shown in Figure 12.1.

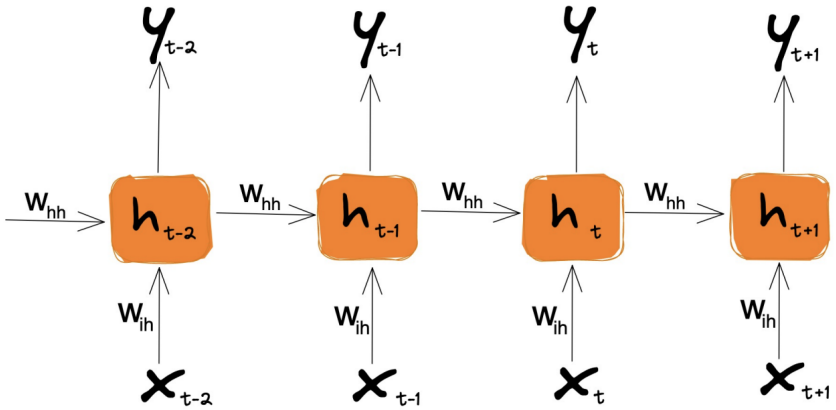


Figure 1: A recurrent neural network

A RNN cell consists of a input vector x_t, weight matrices W_{ih}, W_{hh}, W_{hy}, and performs forward pass as follows:

$$H_t = \tanh(W_{ih}x_t + W_{hh}H_{t-1} + b_h) \tag{49}$$
$$y_t = g(W_{hy}H_t + b_y), \tag{50}$$

where $g(\cdot)$ is an activation function. Typically, W_{hh} and W_{ih} are concatenated into W, and multiplied with concatenating of H_{t-1} and x_t.

During **training**, a training sequence (e.g. a word), is input to the network, word by word, to predict the next character. During **testing**, we sample from the model to synthesize new text, by using the output at cell t_0 as input to the RNN cell at t_1.

12.2 Backpropagation through Time (BPTT)

RNNs are trained using backpropagation through time (BPTT). The network is unrolled, and the RNNs is treated like standard neural network, and backpropagation is applied. Note that gradients at time step k receives gradients from all the times steps t where $t > k$.

12.3 Truncated BPTT

For very long time series, BPTT will cause memory issues, since a lot of gradients need to be calculated and stored. A solution is to stop at some point and update the weights as if we are done – this is called **truncated BPTT**. Results in faster parameter parameters, and is more memory efficient, but does not capture longer dependencies than the truncated length.

12.4 Exploding and Vanishing Gradients

During backpropagation, very long gradient chains can arise due to the nature of RNN architecture. Thus, it is essential to look at the activation function's derivative at any point n in the network. Based on its value, it may lead to two cases:

$$g'(z) < 1 \quad \Rightarrow \quad \text{Vanishing gradients} \tag{51}$$
$$g'(z) > 1 \quad \Rightarrow \quad \text{Exploding gradients} \tag{52}$$

For g =ReLu, the derivative has no upper bound, leading to exploding. For g =Logits < 1, gradients may vanish. For $g = \sigma(z)$ or $g = \tanh$, derivatives have upper bounds, but can lead to vanishing gradients. To avoid this we can use **gradient clipping**, either by clipping-by-value or clipping-by-norm. This clips gradients if they are larger than a threshold. Another way of preventing vanishing and exploding gradients is to initialize the weight matrix as **orthogonal matrices**. To prevent vanishing gradients, we can also use **residual connections**, proper **weight initialization**, **batch normalization**, and/or **input normalization**. For RNNs, we may also use cells such as **GRU** and **LSTM**.

12.5 LSTM

A more sophisticated way to **prevent vanishing gradients** and **preserve long-term dependencies** is to use LSTM (Long Short Term Memory) networks. Introduce a cell state C, serving as memory. Other new components are **gates**, which modify the cell state, and decides how much of the old cell state to keep.

The LSTM cell contains the following gates:

The **input gate**, controlling what fraction of the new memory is added to new one. First a new candidate is generated:

$$\hat{C} = \tanh(W_c[h_{t-1}x_t] + b_c) \tag{53}$$

Then, the input gate determines the fraction of $\hat{C}$ to be used:

$$i_t = \sigma(W_i[h_{t-1}x_t] + b_i) \tag{54}$$

The **output gate** decides how much of a output candidate, defined as $\hat{h}_t = \tanh(C_t)$, where C_t is the cell state, we are going to output, calculated as:

$$o_t = \sigma(W_o[h_{t-1}x_t] + b_o) \tag{55}$$

The **output of the LSTM cell** is then

$$h_t = o_t \times \hat{h}_t, \tag{56}$$

updating the **hidden state**, and

$$C_t = f_t \times C_{t-1} + i_t \times \hat{C}, \tag{57}$$

updating the **cell state**.

12.6 GRU

Gated recurrent units (GRUs) is a variant of LSTM, but simpler and smaller, i.e. less parameters. It *combines* the **input gate** and **forget gate** from LSTM into a single update gate.

12.7 Bidirectional RNNs

For some applications, it helps looking into the **future**. This can be achieved using bidirectional RNNs. Introduces hidden blocks in each direction. Examples include speech recognition and handwriting recognition.

12.8 Image Captioning – CNN + RNN

CNNs can be combined with a RNN to generate descriptive text (captioning) for an image. Idea: Input image x is sent thought a CNN, where the last fully connected layers and Softmax has been removed. The convoluted image works as input to the first hidden

layer of the RNN, along with the caption (during **training**) or by using the RNN output as input at time $t+1$ (during **testing**).

13 Object Detection

The goal in object detection is to both perform **image classification** and predict where in the image the object is located by drawing a **bounding box** around it.

13.1 Single-instance detection and localization

For a single-instance object detection, we can start by considering a normal image classification network. To extend this to object detection, we need to **adjust the output vector** and the **loss function**.

Given input image x, with possible classes $c = c_1, c_2, \ldots, c_K$, and prediction vector $\mathbf{y}$. We can extend this to object detection by adding a class for background (c_0) determining no object, and values that describe the bounding box: center (b_x, b_y), width (b_w) and height (b_h), resulting in the output vector:

$$\mathbf{y} = (c_0, c_1, c_2, \ldots, c_K, b_x, b_y, b_h, b_w). \tag{58}$$

Furthermore, the total loss function will consist of two separate losses; a cross entropy loss for the classes, and L_2 loss for the bounding box, resulting in the total loss:

$$L(y, \hat{y}) = -\sum_{i=0}^{K} c_i \log \hat{c}_i + [c_0 \neq 1] \sum_{i \in \{x,y,h,w\}} (b_i - \hat{b}_i)^2 \tag{59}$$

Note that the second term is only included if there is an object present.

13.2 Object detection of Multiple Objects

When multiple objects are present there are several methods for detecting them.

13.2.1 Sliding window approach

Slides windows of different sizes across image, applying image classifier at each location. Slow for CNN classifiers, but OK for cheap classification methods.

13.2.2 Two stage detectors

The two stage detector method is split into two stages. First, the method **generates multiple candidates** – bounding boxes, typically with the use of anchors. Then throw

away candidates without object. Second, each **candidate is processed** independently by classification and adjustment of the bounding box.

13.2.3 One stage detectors

Pass the image though a CNN, and on the final feature map perform sliding window. At each window: predict object and bounding box per anchor / prior. Can also be done **anchor-free**, by directly predicting the coordinates of the bounding box and/or centre of the object.

13.3 Anchors

Anchor boxes allow multiple objects to have the same center location, however, they may differ in bounding box accuracy. Boxes based on anchors are also known as *priors* or *default boxes*. Using anchors reduces the number of candidates, potentially speeding up the process of object detection.

13.4 Common Problems in Object Detection

13.4.1 Varying Object Sizes

Large and small objects can be present, causing problems during object detection. As a results we may use **single-shot multibox detector**, which performs classification ad object detection for multiple sizes of the input image, after processed by a CNN. An alternative is using **feature-pyramid networks** – where predictions are performed on varying sizes of the feature maps.

13.4.2 Overlapping Predictions

In case of overlapping, use **non-maximum supression** (NMS), based on the **Intersection over Union** (IoU), defined as

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} \tag{60}$$

A good indication is that IoU $\rightarrow 1$.

13.4.3 Many Background Predictions

In case the anchor-boxes come without a match one can use **hard negative mining**, which selects the most difficult background patches when computing loss. Alternatively, use **focal loss**. Focal loss differs from ordinary cross-entropy loss by focusing training on a sparse set of hard examples.

The **focal loss** function is defined as:

$$FL(p_t) = -(1 - p_t)^{\gamma} \log(p_t), \tag{61}$$

where

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{else} \end{cases} \tag{62}$$

13.5 Performance and Evaluation Metrics - Object detection

Compute precision, recall and AP per class, and mAP over all classes. The average precision per class can be denoted AP_K for class K.

14 Image Segmentation

14.1 Semantic Segmentation

In semantic segmentation, the goal is to predict **object classes** and **background classes**. In this case, every pixel is classified, and the model should differentiate between classes, but not between multiple instances of same class. Example: all cats are detected (colored) as one.

14.1.1 Approaches in Semantic Segmentation

First, we can consider **sliding windows** – classifying each pixel. However, this is very computationally expensive, and it lacks global context. Second, we can consider CNN **without downsampling**. However, the receptive field is small, it becomes expensive for large images, and we cannot use pre-trained networks.

The ideal scenario is to have large receptive field, to capture large areas for context, and using low res information to capture fine details.

14.1.2 Capturing Global Context

Capturing global context means getting information from the entire image. One solution is to **downsample feature maps** using max or average pooling and/or stride > 1. Another solution is to use **dilated convolution**, also called atrous convolution. An improved solution is to **encode** the input vector through a CNN, before **decoding** it by upsampling/unpooling/"deconvolution" of the encoded vector back to the initial image size.

14.2 Upsampling and "Deconvolution

Through upsampling of an image, we get back to the image-level resolution; it's initial size. Examples of upsampling techniques include:

- Nearest Neighbour upsampling – Copies neighbour
- Bi-linear upsampling
- Bi-cubic upsampling
- Transposed Convolution / Fractionally Strided Convolution / "Deconvolution"
- Unpooling – Stores indices of Max pooling, and used to place unpooled output, pad with zeros.

Specifically, in **transposed convolution**, parameters of kernel K are learned to enlarge the input. Stride and padding also affects output size. Final feature map is a sum of a number of "convolutions".

14.3 Dilated / Atreus Convolution

The output is given by the update formula:

$$y[i] = \sum_k x[i + r, k]w[k], \tag{63}$$

where r is the dilation rate. Standard convolution is the case where $r = 1$. Dilated convolution **replaces downsampling** layers.

14.4 Instance Segmentation

In instance segmentation, the goal is to predict **object classes** and **object instances**. Here, only pixels belonging to objects are labeled, but different instances of the same class are labeled differently. Approaches include performing object detection, generating a mask for each object in parallel, and using Mask R-CNN.

14.4.1 Mark R-CNN

Extends Faster R-CNN by predicting mask in parallel to box offset and class – thus a mask is predicted for each class for each region of interest.

14.5 Performance Metrics in Image Segmentation

Main metrics include **mean IoU** and **mask mAP**. The most popular choice is mean IoU, defined as average IoU over all classes – used for **semantic segmentation** The mask mAP metric is mainly used for **instance segmentation**.

Other metrics include *pixel accuracy, mean pixel accuracy* and *dice coefficient*.

14.6 Loss functions in Image Segmentation

For semantic segmentation: **Cross entropy** for each pixel. For instance segmentation: **sum of three losses**; classification loss (cross entropy), bounding box loss (regression) and mask loss (cross entropy per pixel).

14.7 Panoptic Segmentation (Definition)

In panoptic segmentation, the goal is to do both **semantic and instance segmentation**.

15 Explainability in Neural Networks

Explainability in neural networks includes a vast amount of explanations and questions. This includes understanding the model (**model interpretation**), working with explainable models (**training explainable models**), and understanding model predictions (**decision interpretation)**.

15.1 Model interpretation

Interpreting the model can include creating a distance map between feature maps (**t-SNE**) or trying to understand what input affects a layer the most.

15.1.1 t-SNE

The t-SNE method defines low-dimension representations of high-dimensional feature maps. The goal is to cluster points in low-dimension based on which represent close feature maps in high-dimension; e.g. by projecting the feature maps down into two dimensions.

Idea of **t-SNE**: given samples i, j with features u_i, u_j, we can find the probability p_{ij} which is a model of the **interaction strength** between the two samples. Then, in the lower dimension for our low-dimensional representatives y_i, y_j, we can define a model of interaction strength $q_{ij} = q(y_i, y_j)$, and tune the parameters of $q(\cdot)$ until

$$q_{ij} \approx p_{ij} \tag{64}$$

Optimization is performed using the Kullback-Leibler-Divergence.

Alternatives is PCA projections, and Isomap methods.

15.1.2 Nearest examples

In this method, the idea is to select a sample x, and feature map $h(x)$, and find the closest examples in a test set. Then sort x_i according to the norm:

$$\|h(x_i) - h(x)\| \tag{65}$$

This will try to explain similarities between x_i and a sample x defined by the chosen feature map.

15.2 Decision interpretation

One possible approach to explaining predictions on a single sample is by **decomposition**.

15.2.1 Shapley values

One decomposition approach is using **Shapley values** and looking at a subset of features from some point to be explained, x. Shapley value approach attempts to explain why the model produces the output / feature map that it does based on the input x.

15.2.2 Linearisation methods

The idea of linearisation methods is to have an input sample, $x = (x_1, \ldots, x_D)$, a classifier $f(x)$, and try to understand **what parts** of x are important for the prediction of $f(x)$. There are several proposed solutions, mostly involving gradients:

The **gradient** itself does not explain which pixels are most contributing to the prediction, but it does show which pixels are most sensitive to change the prediction.

The **Gradient×Input** method aims to use Taylor linearisation for a point x_0 orthogonal to the gradient in the point x to be explained:

$$f(x) \approx f(x_0) + \nabla f(x) \cdot x, \tag{66}$$

where the term $\nabla f(x) \cdot x_0 = 0$ due to orthononality. However, this method can be noise in ReLU networks due to gradient shattering

The **integrated gradient** method is an improvement to the gradient × input method, avoiding gradient shattering through averaging gradients. However, it quickly gets slow (and better) when hundreds of points are used.

The **Grad-CAM** method is a heuristic variation of the gradient × input method; applied to the feature map space. The feature maps we are interested in are in the last layer, where the non-linear activation has been removed to preserve linearity. Grad-CAM replaces the gradient in gradient × input with a spatially averaged version – thus averaging over all feature maps using a weighted sum, as shown in Figure 15.2.2.

The **guided backpropagation** method is a heuristic which cancels out parts of the backpropagated terms, and is often combined with other explainability methods. Given a activation layer with function $g(z)$, perform backpropagation, but **zero out** incoming gradient at a layer if:

- Input to activation is negative, $z < 0$
- Arriving gradient is negative, $\nabla L_g(z) < 0$

This means we only want activating neurons, and only consider gradients which increase prediction. Thus, the method only considers activating signals and gradients. Generates clean heatmaps, high resolution and easy to implement with ReLUs, but is complex to implement for non-ReLU activations, and lacks sensitivity to explanation for different classes.

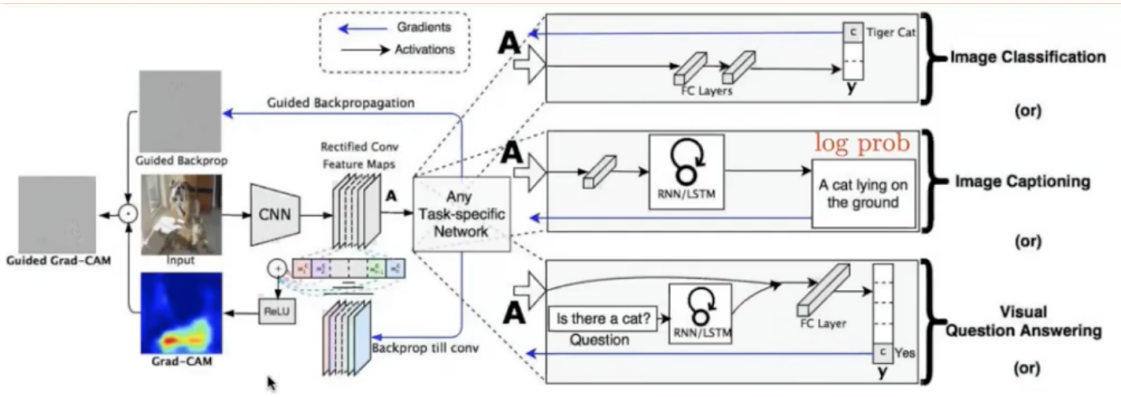


Figure 2: Visualization of Grad-CAM and guided backpropagation

15.2.3 Occlusion

By occluding an input x with classifier $f(x)$, we **occlude a subset** of x to generate occluded sample $\tilde{x}$, and measure the sensitivity $r_S(x) = f(x) - f(\tilde{x})$ By investigating multiple subsets, we can find the most sensitive subsets, i.e. the parts of the image which contributes the most to the prediction. This is a simple method, black-box, and

is a direct measurement of sensitivity. However, the approach is really slow, and we also need to decide occlusion region and size.

15.3 Layer-wise Relevance Propagation (LRP)

The LRP algorithm explains a classifier's prediction specific to a given point x by generating relevance scores and linking it to important components of the input, based on the structure of the learned model.

16 Generative Adversarial Networks (GANs)

16.1 General Idea

The goal of generative adversarial networks (GANs) is to generate new data with the same statistics as the training set. This is performed by using a "counterfeiter" and a "detective", or in more technical terms: by sending a random noise vector through a **Generator**, followed by a classifier called the **Discriminator**. Hence, GANs work in two steps: learning a model G that generates new samples from a random noise vector, and training of G should produce outputs $x = G(z)$ that are **similar to images in a given dataset**.

Similar in the case of GANs implies that given two distributions P_{data} and P_{GAN}, representing the sampled and generated data, we aim to make

$$P_{GAN} \approx P_{data} \tag{67}$$

16.2 Generator

The generator G takes as input a vector z, randomly sampled from a random distribution, and outputs $x = G(z)$. The image x is a full sized image, ready to be discriminated. We start with a low-resolution vector, but need to increase the resolution progressively, and possibly decrease the number of channels in the feature map. We have two options: **upsampling** followed by convolutions, or **fractionally strided convolution** known as "deconvolution".

16.3 Discriminator

The discriminator is essentially a quality measure for the generator G. A GAN discriminator $D(x)$ relies on a set of images as input and is a **classifier** with one output, or **encoder** for determining whether the image is fake or real. If $D(x) = 1$, the discriminator things that x is real, as $D(x)$ represents probability.

16.4 Loss functions

Assume pre-trained and fixed $D(x)$. The goal is to make $D(G(z)) \to 1$, hence we can minimize:

$$\min_G \ln[1 - D(G(z))], \tag{68}$$

resulting in the following loss for the generator $G(z)$.

The **generator loss function** is

$$L_G = \min_G \frac{1}{K} \sum_{z_i \in \mathcal{B}} \ln[1 - D(G(z))], \tag{69}$$

for one mini-batch. During training, the discriminator is **not** fixed, to avoid *mode collapse*.

For the discriminator, we want $D(x) \approx 1$, and $G(z)$ should be classified by the discriminator as not coming from D_n when $D(G(z)) \approx 0$, meaning we want to minimize

$$\min_D -\ln D(x_i) - \ln[1 - D(G(z_i))], \tag{70}$$

where $x_i \in D_n$, resulting in the following loss for the discriminator $D(x)$.

The **discriminator loss function** is

$$L_D = \frac{1}{b} \sum_{x_i \in D_n}^{b} -\ln D(x_i) + \frac{1}{b} \sum_{z_i \in P_z}^{b} -\ln[1 - D(G(z_i))]. \tag{71}$$

A GAN using the loss functions above is called a **non-saturating GAN**.

16.4.1 Mode collapse

In the case of mode collapse, the generator distribution P_{GAN} only covers a small part of the space represented by the test distribution, resulting in

$$P_{GAN}(A) \ll P_{test}(A), \qquad A \subset \mathbb{R}^d \tag{72}$$

16.5 Problems in GANs

Generator quality vs diversity: Can come in conflict, e.g. when using a fixed discriminator and no tricks on the generator. An alternative is to train the discriminator and generator **jointly**.

16.6 Training a GAN

Step 1: Train discriminator with fixed generator.

Fix generator weights, generate input and label as fake class.

Train discriminator weights, altering between real and fake samples

Cross-entropy loss function for classification

Step 2: Train generator with fixed discriminator.

Fix discriminator weights.

Train generator weights in order to fool the discriminator.

16.7 Wasserstein GAN

An alternative loss in GANs is the Wasserstain distance. We want to redistribute two samples / distributions, using a redistribution function $\gamma(x_i, x_k)$ such that $\mu(x_i) = P_{data}$ is similar to $\tau(x_k) = P_{GAN}$. An improved loss function for the the generator and discriminator is then as follows.

The **generator loss** for a Wasserstein GAN is:

$$\min_{w(G)} L(G), \quad L(G) = -\frac{1}{b}\sum_{z_i}^{b} f(G(z_i)) \tag{73}$$

The **discriminator loss** for a Wasserstein GAN is:

$$\min_{w(D):D\in Lip_1} -\frac{1}{b}\sum_{x_i}^{b} D(x_i)) + \frac{1}{b}\sum_{z_i}^{b} D(G(z_i)), \tag{74}$$

where Lip_1 implies that we need $f \in Lip_1$, ensuring Lipschitz property.

16.8 Metrics in GANs

Quality evaluation in GAN is not a fully solved problem. One measure we can use is the **KL-Divergence**, also known as **relative entropy** between two distributions. The KL-Divergence measure how different the distributions are.

A second measure is the **inception score**, which builds on our intuition that images generated in GANs should relate to a class humans can relate to. Also, with a variety of images, we expect a variety of classes. Is measured using a Softmax probability $p(y|x)$, which should peak for a good generator concentrated on one class, but have low entropy

since it is good at distinguishing objects. However, $p(y)$ is the marginal distribution, created over many classes and should be flat, i.e. large entropy.

The inception score, $IS(G)$ is defined as:

$$IS(G) = e^{\mathbb{E}_{x\sim G} D_{KL}(p(y|x)\|p(y)))}. \tag{75}$$

May be unreliable because it relies on InceptionV3 network. It is also weak at detecting mode collapse.

A third measure is the **Frechet inception distance**, defined in the feature map space, not in pixel space. Hence it can capture high level semantic similarities and can detect model collapse, but relies on implementation of InceptionV3.

A fourth measure is the **perceptual path length** (PPL), considering the latent space and its "curvature". A less curved latent space should result in perceptually smoother transition. Uses the $slerp(x, y, t)$ function, and computes the expectation over random draws of two random input codes z_1, z_2. Detects mode collapse, and correlates to human rating, but exploits properties of current GAN architecture, which may change in the future.

17 Applications of GANs

17.1 Progressive Growing

The idea of progressing growing is to train a sequence of increasingly harder problem, at increasing resolutions, by gradually **fading in** additional layers. The method upscales the neural network after finished training at one resolution. One drawback is that the upscaling does not add finer details.

In **progressing growing**, a model is trained at a certain resolution, with output y_i. Then, a module is added on top of the **generator**, including upscaling and perform convolution as follows:

$$z_{i+1} = upscaleNN(y_i) \tag{76}$$
$$y_{i+1} = \alpha conv3 \times 3(z_{i+1}) + (1-\alpha)z_{i+1}, \tag{77}$$

where α is initially 0, and then increase until $\alpha = 1$, gradually adding more and more noise to the image.
For the **discriminator**, add average pooling layers and convolution:

$$z_{i+1} = avgpool2(y_i) \tag{78}$$
$$y_{i+1} = \alpha avgpool2(conv3 \times 3(y_i)) + (1-\alpha)z_{i+1}, \tag{79}$$

to decrease the resolution and scale down.

Additional concepts include normalizing feature maps, and re-scaling of the weights.

17.2 Application: Morphing Images

Given points x_1 and x_2, find z_1, z_2 such that

$$z_1 = \arg\min_z \|f(x_1) - f(G(z))\| \tag{80}$$
$$z_2 = \arg\min_z \|f(x_2) - f(G(z))\|, \tag{81}$$

and interpolate K points between z_1 and z_2 to create a morphed image.

17.3 Application: Cross Domain mappings - To subdomain

An example is mapping faces to emojis without ground truth labels for what to map on what. The generator G is a decoder-encoder, with pretrained encoder, see Figure 17.3 Loss is a 3-class GAN loss considering (1) original data, (2) fake face, (3) fake emoji.

17.4 Application: Cross Domain mappings - Via CycleGAN

An example is using the generator G to map from horses to zebras, and one generator F to map from zebras to horses.

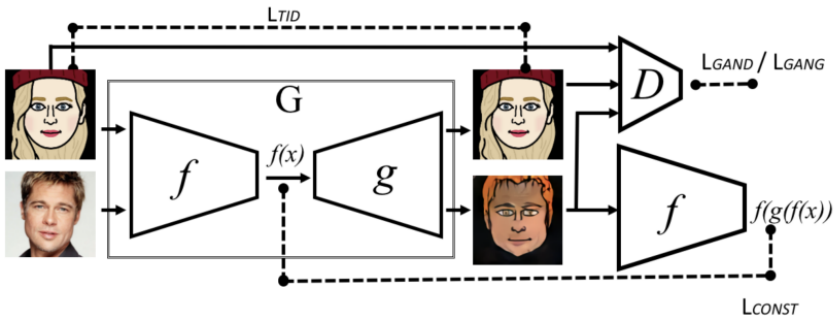


Figure 3: A domain transfer network.

Cross domain mapping with CycleGAN results in a generator G and one generator F such that:

G: Horses $\rightarrow$ Zebras

F: Zebras $\rightarrow$ Horses

F(G): Zebras $\rightarrow$ Zebras

G(F): Horses $\rightarrow$ Horses

An additional term is added to the total loss:

$$\|F(G(x) - x\|_1, \tag{82}$$

implying that an input x mapped by F and back by G should be approximately x.

17.5 StyleGan

Idea is to **add random noise** to every feature map, which gets smoothed out as it passes other layers. Second idea is to use adaptive instance normalization. Third idea is to use progressive growing as explained above.

Adaptive instance normalization of some input x, is defined as:

$$AI(x, y) = y_s \frac{x - \mu_x}{\sigma_x} + y_b, \tag{83}$$

where the style vectors y_s and y_b are **not** trainable. Instead they are inputs from a trainable network. In contrast to batch normalization, the data statistics will be adjusting based on the instance, not batch. Thus, the statistics are computed across the spatial dimensions independently for each channel and each sample.

17.6 Training GANs on small datasets

17.6.1 Overfitting in GANs

A first problem of GANs with small datasets is **overfitting**. Overfitting in GANs can be measured by comparing **discriminator scores** on training data, generated data, and withheld real images. Solution is to use data augmentation for G and D, with a probability $1 - p \in [0, 1]$. But how is p selected?

17.6.2 Measuring Overfitting in GANs

To measure the overfitting, we can define to measures:

$$r_v = \frac{E[D_{train} - E[D_{val}]}{E[D_{train}] - E[D_{generated}]} \tag{84}$$

$$e_t = E_{x \sim TrainData}[sign(D_{train}(x)], \tag{85}$$

where $0 \Rightarrow$ no overfitting, and $1 \Rightarrow$ bad overfitting. The idea is to keep r_v and r_t approximately constant by initializing $p = 0$, and adjusting p every 4 mini-batches. Keep $r_t \approx 0.6$.

17.6.3 Transfer Learning in GANs

We can apply transfer learning for the discriminator by freezing only the layers for the 3-4 highest resolution.

17.6.4 Inpainting

The idea of inpainting is to **fill missing regions** of an image such that the image is complete and looks realistic. To fix this, we can use a discriminator D on the **whole image**. One example architecture can be in two stages:

Stage 1: Coarse reconstruction of missing patch

Stage 2: Apply reconstruction loss and two discriminators (local and global)

In stage 1, the training loss is a **weighted pixel-wise reconstruction loss**:

$$L(x, f(x)) = \|(f(h(x)) - x) \times \gamma(h(x))\|_1, \tag{86}$$

where the weight γ is defined as

$$\gamma(\text{pixel}) = \begin{cases} 1 & \text{If pixel is not white} \\ \to 0 & \text{If pixel is white (Inside patch)} \end{cases} \tag{87}$$

In stage 2, the training loss is the **sum of three losses**:

- Weighted pixel-wise reconstruction
- Discriminator on patch (local)
- Discriminator on whole image (global)