Machine Perception

Cristian Perez Jensen

April 3, 2024

Contents

List of symbols

$\doteq$	Equality by definition
$\approx$	Approximate equality
$\propto$	Proportional to
$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$i:j$	Set of natural numbers between i and j. *I.e.*, $\{i, i+1, \ldots, j\}$
$\mathbb{1}\{\text{predicate}\}$	Indicator function (1 if predicate is true, otherwise 0)
$\boldsymbol{v} \in \mathbb{R}^n$	n-dimensional vector
$\boldsymbol{M} \in \mathbb{R}^{m \times n}$	$m \times n$ matrix
$\mathsf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$	Tensor
$\boldsymbol{M}^\top$	Transpose of matrix $\boldsymbol{M}$
$\boldsymbol{M}^{-1}$	Inverse of matrix $\boldsymbol{M}$
$\det(\boldsymbol{M})$	Determinant of $\boldsymbol{M}$
$\frac{\mathrm{d}}{\mathrm{d}x} f(x)$	Ordinary derivative of $f(x)$ w.r.t. x at point $x \in \mathbb{R}$
$\frac{\partial}{\partial x} f(\boldsymbol{x})$	Partial derivative of $f(\boldsymbol{x})$ w.r.t. x at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^n$	Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{D}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^{n \times m}$	Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^m$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}^2_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^{n \times n}$	Hessian of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\theta} \in \Theta$	Parametrization of a model, where Θ is a compact subset of $\mathbb{R}^K$
$\mathcal{X}$	Input space
$\mathcal{Y}$	Output space
$\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$	Labeled training data

1 Neural networks

1.1 Multi-layer perceptron

The original perceptron [Rosenblatt, 1958] was a single layer perceptron with the following non-linearity,

$$\sigma(x) \doteq \mathbb{1}\{x > 0\}.$$

The classification of a single point can then be written as

$$\hat{y} = \mathbb{1}\{\boldsymbol{w}^\top \boldsymbol{x} > 0\}.$$

The learning algorithm then iteratively updates the weights for a data point that was classified incorrectly,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \underbrace{(y_i - \hat{y}_i)}_{\text{residual}} \boldsymbol{x}_i,$$

where η is the learning rate. If the data is linearly separable, the perceptron converges in finite time.

The problem with the single-layer perceptron was that it could not solve the XOR problem; see Figure 2. This can be solved by introducing hidden layers,

$$\hat{y} = \sigma(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \cdots \sigma(\boldsymbol{W}_1 \boldsymbol{x}))).$$

We call this architecture a multi-layer perceptron (MLP); see Figure 3. We then want to estimate the parameters $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_k\}$, using an optimization algorithm such as gradient descent, which we call "learning".

1.2 Loss functions

We need an objective to optimize for. We typically call this objective function the loss function, which we minimize. In classification, we typically optimize the maximum likelihood estimate (MLE),

$$\begin{aligned}
\operatorname*{argmax}_{\boldsymbol{\theta}} p(\mathcal{D} \mid \boldsymbol{\theta}) &\overset{\text{iid}}{=} \operatorname*{argmax}_{\boldsymbol{\theta}} \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\sum_{i=1}^{n} \log p(y_i \mid \boldsymbol{\theta}).
\end{aligned}$$

If the model predicts the parameters of a Bernoulli distribution, MLE is equivalent to binary cross-entropy,

$$\begin{aligned}
\ell(\boldsymbol{\theta}) &= -\log \mathrm{Ber}(y_i \mid \hat{y}_i \doteq f(\boldsymbol{x}_i \mid \boldsymbol{\theta})) \\
&= -\log \hat{y}_i^{y_i} (1 - \hat{y}_i)^{1 - y_i} \\
&= -y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i).
\end{aligned}$$

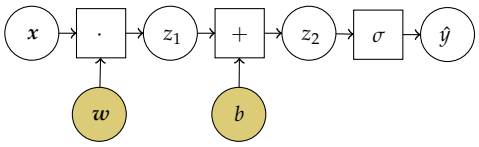

Figure 1. Computation graph of a perceptron [Rosenblatt, 1958], where $\sigma(x) = \mathbb{1}\{x > 0\}$.

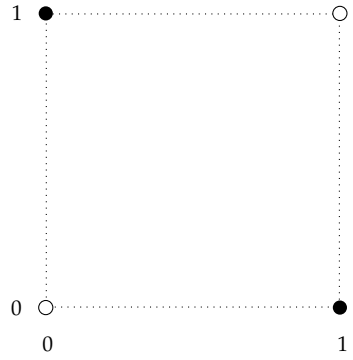

Figure 2. XOR problem. As can be seen, the data is not linearly separable, and thus not solvable by the perceptron.

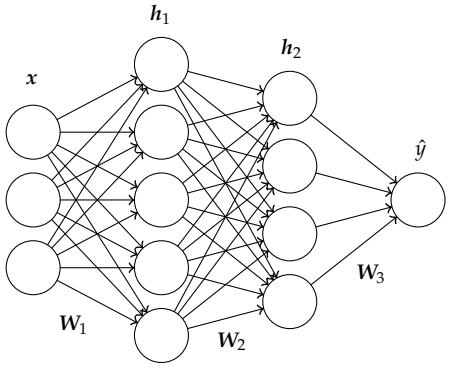

Figure 3. Example multi-layer perceptron architecture.

f is a model that outputs the Bernoulli parameter.

If we the choose the model to be Gaussian, we end up minimizing the mean-squared error. Furthermore, the Laplacian distribution yields minimizing the ℓ_1 norm.

If we have prior information about the weights, we could also optimize for the maximum a posteriori (MAP),

$$\begin{aligned}
\operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathcal{D}) &= \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) p(\mathcal{D} \mid \boldsymbol{\theta}) \\
&\overset{\text{iid}}{=} \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log \left(p(\boldsymbol{\theta}) \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) \right) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log p(\boldsymbol{\theta}) - \sum_{i=1}^{n} \log p(y_i \mid \boldsymbol{\theta})
\end{aligned}$$

Note that MAP and MLE are equivalent if $p(\boldsymbol{\theta})$ is uniform over the domain of weights. Assuming a Gaussian prior distribution over $\boldsymbol{\theta}$, MAP yields Ridge regression.

1.3 *Backpropagation*

Typically, we cannot find the optimal parameters $\boldsymbol{\theta}^\star$ in closed form, so we must use an optimization algorithm. Optimization algorithms, such as gradient descent, typically require computing the gradient w.r.t. the parameters. Backpropagation is an algorithm for computing the gradient of any function, given that we have access to the derivatives of the primitive functions that make up the function.[1] It then computes the gradient by making use of dynamic programming, the chain rule, and sum rule.

[1] For example, to compute the gradient of $f(\boldsymbol{x}, \boldsymbol{y}) = \sigma(\boldsymbol{x}^\top \boldsymbol{y})$, we would need access to $\frac{\mathrm{d}}{\mathrm{d}x}\sigma(x)$, $\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{x}^\top \boldsymbol{y}$, and $\frac{\partial}{\partial \boldsymbol{y}} \boldsymbol{x}^\top \boldsymbol{y}$.

Gradient descent iteratively updates the parameters by the following,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}),$$

until the gradient is small.

1.4 *Activation functions*

In MLPs, the activation function should be non-linear, or the resulting MLP is just an affine mapping with extra steps. This is because the product of affine mappings are themselves affine mappings.

1.5 *Universal approximation theorem*

Theorem 1 (Universal approximation theorem)**.** Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a non-constant, bounded, and continuous activation function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$ and the space of real-valued function on I_m is denoted by $\mathcal{C}(I_m)$.

Let $f \in \mathcal{C}(I_m)$ be any function in the hypercube. Let $\epsilon > 0, N \in \mathbb{N}, v_i, b_i \in \mathbb{R}, \boldsymbol{w}_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, then

$$f(\boldsymbol{x}) \approx g(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \sigma(\boldsymbol{w}_i^\top \boldsymbol{x} + b_i),$$

where $|f(\boldsymbol{x}) - g(\boldsymbol{x})| < \epsilon$ for all $\boldsymbol{x} \in I_m$.

The universal approximation theorem holds for any single hidden layer network. However, this hidden layer may need to have infinite width to approximate f. In practice, deeper networks work better than wider networks.

2 Convolutional neural networks

When dealing with high-dimensional data, such as images, it is not practical to work with MLPs, because the amount of parameters would explode in size.[2] By making use of the locality of images, we can drastically decrease the number of parameters.

[2] Mapping a $256 \times 256 \times 3$ input image to a 1-dimensional output would already require nearly 2 million parameters.

2.1 Convolution

The *correlation* operator takes a filter $\mathbf{K}$, moves it along the entire image, and outputs the patch-wise multiplication, for each patch of the same size as the filter. It is defined as follows,

$$(\mathbf{K} \star \mathbf{I})[i,j] = \sum_{m=-k}^{k} \sum_{n=-k}^{k} \mathbf{K}[m,n]\mathbf{I}[i+m, j+n].$$

The *convolution* operator is very similar. The only difference is that the kernel is mirrored in a convolution,

$$(\mathbf{K} * \mathbf{I})[i,j] = \sum_{m=-k}^{k} \sum_{n=-k}^{k} \mathbf{K}[m,n]\mathbf{I}[i-m, i-n].$$

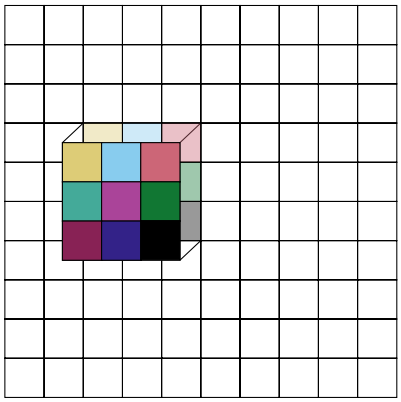

Figure 4. Illustration of applying a correlation to a pixel.

Theoretically, the convolution operator is more useful, because it is commutative.[3] In practice with neural networks, it does not matter, since the weights will just be updated to be the same, except that they are mirrored. Thus, we will only be referring to the convolution from now on.

[3] $\mathbf{I} * \mathbf{K} = \mathbf{K} * \mathbf{I}$, but $\mathbf{I} \star \mathbf{K} \neq \mathbf{K} \star \mathbf{I}$.

A convolution operator C is a linear, shift-equivariant transformation, *i.e.*,

$$C(\alpha \boldsymbol{x} + \beta) = \alpha C(\boldsymbol{x}) + \beta$$

Linearity.

$$T_t(C(\boldsymbol{x})) = C(T_t(\boldsymbol{x})).$$

Translation equivariant.

Since convolutions are linear, discrete convolutions can be implemented using matrix multiplication,

$$\mathbf{K} * \mathbf{I} = \begin{bmatrix} k_1 & 0 & 0 & \cdots & 0 \\ k_2 & k_1 & 0 & \cdots & 0 \\ k_3 & k_2 & k_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & k_m \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ \vdots \\ I_n \end{bmatrix}.$$

2.2 Convolutional neural network

A convolutional neural network (CNN) is composed of a sequence of *convolutional layers* and *pooling layers*, followed by a final *dense layer* (MLP).

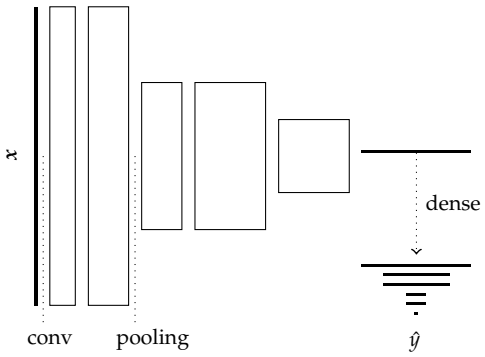

Figure 5. Example schematic of a CNN architecture.

Convolutional layer. A convolutional layer applies a filter $\boldsymbol{W}$ to an input image $\boldsymbol{Z}$ by convolution. This filter is learned. Hence, we need to derive its derivative. We will focus on the single filter case for simplicity, whose forward pass is computed by

$$z^{(\ell)}[i,j] = \sum_{m=-k}^{k} \sum_{n=-k}^{k} w^{(\ell)}[m,n] z^{(\ell-1)}[i-m, j-n] + b.$$

We can express the derivative of the cost function $\mathcal{L}$ w.r.t. the output of the $(\ell-1)$-th layer as the following,

$$\begin{aligned}
\delta^{(\ell-1)}[i,j] &= \frac{\partial \mathcal{L}}{\partial z^{(\ell-1)}[i,j]} \\
&= \sum_{i'} \sum_{j'} \frac{\partial \mathcal{L}}{\partial z^{(\ell)}[i',j']} \frac{\partial z^{(\ell)}[i',j']}{\partial z^{(\ell-1)}[i,j]} \\
&= \sum_{i'} \sum_{j'} \delta^{(\ell)}[i',j'] \frac{\partial}{\partial z^{(\ell-1)}[i,j]} \sum_{m} \sum_{n} w^{(\ell)}[m,n] z^{(\ell-1)}[i'-m, j'-n] + b \\
&= \sum_{i'} \sum_{j'} \delta^{(\ell)}[i',j'] w^{(\ell)}[i'-i, j'-j].
\end{aligned}$$

From this, we can see that we can compute all values of $\delta^{(\ell-1)}$ by a single convolution,

$$\delta^{(\ell-1)} = \delta^{(\ell)} * \mathrm{Rot}_{180}\left(\boldsymbol{W}^{(\ell)}\right) = \delta^{(\ell)} \star \boldsymbol{W}^{(\ell)}.$$

Using this value, we can compute the derivative w.r.t. the weights, which we need for the parameter update in algorithms such as gradient descent,

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w^{(\ell)}[m,n]} &= \sum_{i} \sum_{j} \frac{\partial \mathcal{L}}{\partial z^{(\ell)}[i,j]} \frac{\partial z^{(\ell)}[i,j]}{\partial w^{\ell}[m,n]} \\
&= \sum_{i} \sum_{j} \delta^{(\ell)}[i,j] \frac{\partial}{\partial w^{(\ell)}[m,n]} \sum_{m'} \sum_{n'} w^{(\ell)}[m',n'] z^{(\ell-1)}[i-m', j-n'] + b \\
&= \sum_{i} \sum_{j} \delta^{(\ell)}[i,j] z^{(\ell-1)}[i-m, j-n].
\end{aligned}$$

Again, this has the form of a convolution, thus we can compute all derivatives of $\boldsymbol{W}^{(\ell)}$ by convolution,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(\ell)}} = \delta^{(\ell)} * \boldsymbol{Z}^{(\ell-1)}.$$

As shown, we can use convolutions for both computing the forward pass, as well as the backward pass. Thus, we first do the forward pass, then compute all $\delta^{(\ell)}$ for all layers ℓ by convolution, and finally we can compute the derivative by convolution as well.

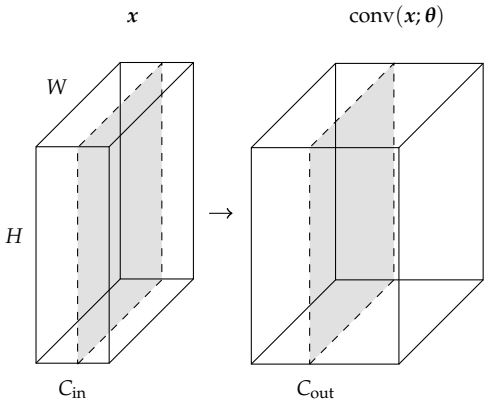

Figure 6. Schematic of a convolutional layer. Each input-output channel pair has its own kernel, so $\boldsymbol{\theta}$ has $K \times K \times C_{\text{in}} \times C_{\text{out}}$ parameters.

Pooling layers. Pooling layers makes the data more manageable. The most common pooling layer is *max pooling*, which outputs the maximum value for each patch. It can be seen as a non-linear convolutional filter, where it simply outputs the maximum value. Usually, pooling is done with a stride, such that the output becomes exponentially smaller; see Figure 7. Because of this, the *receptive field* becomes exponentially larger.

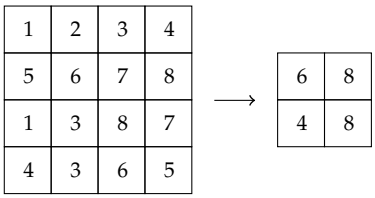

Figure 7. Toy example of max pooling.

The forward pass is computed as follows,

$$z^{(\ell)}[i,j] = \max\Big\{ z^{(\ell)}[i',j'] \;\Big|\; i' \in [si : si+k], j' \in [sj : sj+k] \Big\},$$

where s is the stride and k is the kernel size. Let $[i^\star, j^\star]$ be the indices which corresponded to the maximum value in the forward pass, then we can compute the error propagation in the backward pass by

$$\frac{\partial z^{(\ell)}[i,j]}{\partial z^{(\ell-1)}[i',j']} = \mathbb{1}\{[i',j'] = [i^\star, j^\star]\}.$$

Note that the max pooling layer has no learnable parameters. Hence, the backward pass is only a propagation of the error, and not used for a weight update.

Dense layer. The dense layer is simply a linear layer that maps the final convolutional layer to the network's output. All previous convolutional and pooling layers can be seen as extracting features from the image, while the final dense layer makes the actual prediction.

3 *Fully convolutional neural network*

Semantic segmentation is a computer vision task that involves assigning a semantic class to each pixel in an image. While in image classification, the model must output a single class for the entire image, semantic segmentation requires classifying a class for each pixel individually.

A naive approach would be to apply a single convolutional layer to an image, and then running a classifier on each individual pixel. However, this method is inefficient, because we have to run the classifier $H \times W$ times. Instead, we use the output of convolutional neural networks. A naive approach of using CNNs would be to simply apply n convolutional layers with no downsampling, and then considering the last output as the predicted segmentation map. However, this method is expensive.

In practice, the most common approach is to downsample the features obtained using convolution and pooling layers and then upsample them again. By downsampling, this method is more computationally efficient, has larger receptive field, and suffers less from "The curse of dimensionality". By upsampling, the model produces an output of the same resolution as the input.

3.1 *Upsampling methods*

Nearest neighbor. Nearest neighbor upsampling copies the same value into all corresponding pixels at a higher resolution; see Figure 8.

Bed of nails. Bed of nails upsampling only copies each value once into the output in the top left value, and pads the rest with zero; see Figure 9.

Max unpooling. Max unpooling also uses zero padding, like bed of nails. However, it also remembers the original position of the maximum value before the corresponding max pooling in the downsampling phase. This information is then used to place each element back in the correct position; see Figures 7 and 10.

Transposed convolutions. Transposed convolution is a learned upsampling technique. This layer learns a kernel that is used to produce the terms whose sum will be the final output. Each term is obtained by multiplying all the element of the kernel by the same value of one single input pixel and then inserting the result in the correct position of a matrix of the same size as the output.

3.2 *U-net*

The *U-net* is a FCNN architecture, whose main idea is to combine global and local feature maps by copying corresponding tensors from earlier stages in each upsampling stage; see Figure 11. This allows the network

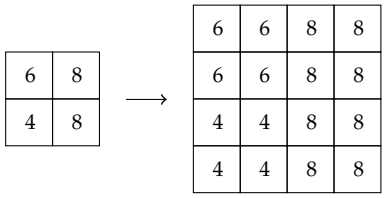

Figure 8. Nearest neighbor upsampling.

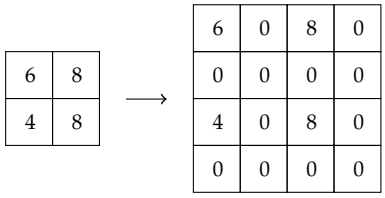

Figure 9. Bed of nails upsampling.

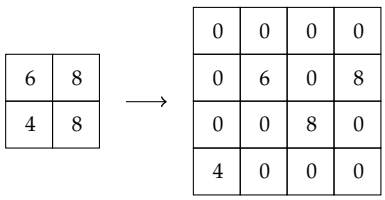

Figure 10. Max unpooling the output of Figure 7.

to capture both local and global context. In each upsampling, the corresponding output from the downsampling phase is appended to the output. The copied tensor can be seen as the "global" information, while the input of the upsampling layer is the "local" information. Combining these allows for more fine-grained outputs.

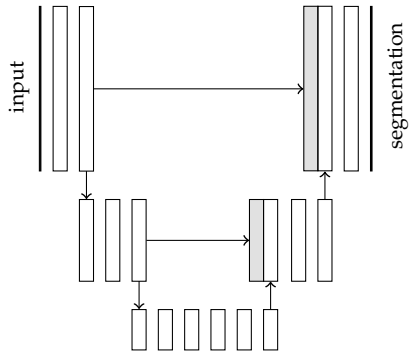

Figure 11. U-net architecture. Down arrows are downsampling layers, up arrows are upsampling layers, and right arrows copy.

4 Recurrent neural networks

Recurrent neural networks (RNN) are a type of neural network that processes sequential data, such as text and video. Unlike traditional neural networks, which take fixed-length inputs, RNNs can take inputs of variable length.[4]

RNNs can have different applications, for example, *one to one*, where at each time step we have one input and one output,[5], *one to many*, where we have one input and we output a sequence of elements,[6] *many to one*, where we have a sequence of inputs and one output,[7] and *many to many*, where we map a sequence to another sequence of a different length.[8]

[4] This is useful for data structures such as text, where the number of words in a text is not fixed.

[5] *E.g.*, part-of-speech tagging.

[6] *E.g.*, image captioning, where the image is the one input, and the caption is a sequence of words.

[7] *E.g.*, sentiment classification, where the input is a text and the output is a single output that determines how positive or negative the text is.

[8] *E.g.*, machine translation, where we map a sentence in one language to a sentence of another.

4.1 Elman RNN

The Elman RNN is characterized by a hidden vector $\boldsymbol{h}^{(t)}$, which forms the state of the network at timestep t. The hidden state is updated at each timestep by combining the previous hidden state with the input,

$$\boldsymbol{h}^{(t)} = \tanh\Big(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)}\Big).$$

We use the tanh activation function, because it is centered at 0.

We can then use the hidden state as a representation of the input up until that timestep. Thus, we can use it as the input to a feed-forward neural network,

$$\hat{\boldsymbol{y}}^{(t)} = \boldsymbol{W}_y \boldsymbol{h}^{(t)}.$$

Then, we can compute the loss function as the sum of each individual loss function,

$$\mathcal{L} \doteq \sum_{t=1}^{T} \ell^{(t)}.$$

We use *backpropagation through time* (BPTT) to compute the gradient of an RNN. This involves first unrolling the RNN; see Figure 12. Then we can compute the gradient by backpropagation on the resulting computational graph,

$$\frac{\partial \ell^{(t)}}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial \ell^{(t)}}{\partial \hat{\boldsymbol{y}}^{(t)}} \frac{\partial \hat{\boldsymbol{y}}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial^{+} \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}},$$

where $\frac{\partial^{+} \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}}$ is the immediate derivative that treats $\boldsymbol{h}^{(k-1)}$ as constant w.r.t. $\boldsymbol{W}$.

Let's only consider the following term of the product,

$$\begin{aligned}
\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} &= \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}} \\
&= \prod_{i=k+1}^{t} \frac{\partial}{\partial \boldsymbol{h}^{(i-1)}} \sigma\Big(\boldsymbol{W}_h \boldsymbol{h}^{(i-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(i)}\Big) \\
&= \prod_{i=k+1}^{t} \boldsymbol{W}_h^{\top} \operatorname{diag}\Big(\sigma'\Big(\boldsymbol{W}_h \boldsymbol{h}^{(i-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(i)}\Big)\Big).
\end{aligned}$$

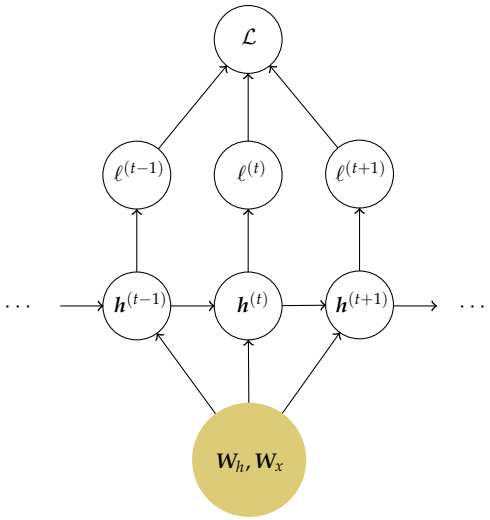

Figure 12. The computational graph of an unrolled recurrent neural network. The inputs $\boldsymbol{x}_{1:T}$ and outputs $\boldsymbol{y}_{1:T}$ are omitted.

Assuming that the norm of the gradient of σ is upper bounded by some $\gamma \in \mathbb{R}$,[9] *i.e.*,

$$\left\| \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} \right) \right) \right\| < \gamma.$$

[9] For example, the gradient of tanh is bounded by 1.

Let λ_1 be the largest eigenvalue of $\boldsymbol{W}_h$, then we have two cases,

1. $\lambda_1 < \frac{1}{\gamma}$. Then we have the following,

$$\left\| \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}} \right\| \leq \|\boldsymbol{W}_h\| \left\| \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} \right) \right) \right\| < \frac{1}{\gamma}\gamma = 1.$$

Triangle inequality and $\|\boldsymbol{W}_h\| = \lambda_1$.

   Let $\eta < 1$ be the upper bound of all gradients between $\boldsymbol{h}^{(i)}$ and $\boldsymbol{h}^{(i-1)}$, then by induction, we have

$$\left\| \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \right\| = \left\| \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}} \right\| < \eta^{t-k}.$$

   This converges to zero, as $t \to \infty$. Thus, we have a *vanishing gradient*;

2. $\lambda_1 > \frac{1}{\gamma}$. Then we have the following,

$$\left\| \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}} \right\| \leq \|\boldsymbol{W}_h\| \left\| \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} \right) \right) \right\| > \frac{1}{\gamma}\gamma = 1.$$

   Using the same logic as in the other case, this yields

$$\left\| \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \right\| = \left\| \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}} \right\| > \eta^{t-k}.$$

   for an upper bound $\eta > 1$. Thus, this diverges to ∞ as $t \to \infty$. Thus, we have a *exploding gradient*.

Thus, we will always have vanishing or exploding gradients when using an Elman RNN. This makes it hard for the architecture to capture long-term dependencies.

4.2 *Long-short term memory*

The *long-short term memory* (LSTM) architecture [Hochreiter and Schmidhuber, 1997] fixes the vanishing gradient problem of the Elman RNN by making sure there is always a path between hidden units, such that errors can always propagate; see Figure 13.

The cell of an LSTM consists of 4 layers, called gates. In particular, these gates have the following instructions,

- f is the *forget gate* and has the role of scaling the old cell state $\boldsymbol{h}^{(t-1)}$. It "decides which information should be forgotten" from the previous cell state, based on the new input $\boldsymbol{x}^{(t)}$;
- i is the *input gate* and has the role of "deciding which values of the cell state $\boldsymbol{c}^{(t)}$ should be updated" at the current time step;

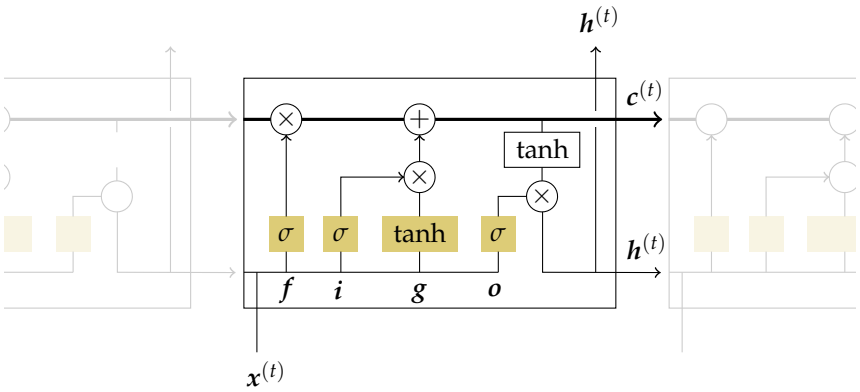

Figure 13. LSTM architecture. The yellow squares are neural networks, and the white squares are point-wise operators. As can be seen, there is an "information highway" that can easily propagate errors at the top, because of the minimal modifications made to it.

- o is the *output gate* and has the role of "deciding which values of the current cell state $\boldsymbol{c}^{(t)}$ should be put in the output of the cell $\boldsymbol{h}^{(t)}$".
- g is the *gate* that decides what to write in the cell state $\boldsymbol{h}^{(t)}$.

We first compute all the gates,

$$\begin{aligned}
\boldsymbol{f}^{(t)} &= \sigma\Big(\boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xf}\boldsymbol{x}^{(t)}\Big) \\
\boldsymbol{i}^{(t)} &= \sigma\Big(\boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xi}\boldsymbol{x}^{(t)}\Big) \\
\boldsymbol{o}^{(t)} &= \sigma\Big(\boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xo}\boldsymbol{x}^{(t)}\Big) \\
\boldsymbol{g}^{(t)} &= \tanh\Big(\boldsymbol{W}_{hg}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xg}\boldsymbol{x}^{(t)}\Big).
\end{aligned}$$

You can also chain multiple LSTM units one after another, which results in this computation being performed multiple times per layer. If this is the case, then we replace $\boldsymbol{x}^{(t)}$ by the hidden vector of the previous unit for all units after the first.

Then, we compute the outputs that are propagated to the next layer,

$$\begin{aligned}
\boldsymbol{c}^{(t)} &= \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \boldsymbol{g}^{(t)} \\
\boldsymbol{h}^{(t)} &= \boldsymbol{o}^{(t)} \odot \tanh\Big(\boldsymbol{c}^{(t)}\Big).
\end{aligned}$$

The addition in the computation of $\boldsymbol{c}^{(t)}$ allows for gradients to directly propagate through $\boldsymbol{c}^{(t-1)} \odot \boldsymbol{f}$. Also, it allows the model to "select" what information should be retained. For example, at a high level in text, it might be helpful to store information such as gender and countries of origin. See Colah's "Understanding LSTM Networks" for more information about the gates.

4.3 *Gradient clipping*

While LSTMs are a great solution to the vanishing gradient problem, we still have the possibility of exploding gradients. This is what *gradient clipping* solves. The idea is to limit the maximum value of the gradient if it surpasses a predetermined threshold. In practice, the gradient descent step gets transformed into the following update rule,

$$\boldsymbol{\theta} \leftarrow \begin{cases} \boldsymbol{\theta} - \gamma \boldsymbol{g} & \|\boldsymbol{g}\| \leq T \\ \boldsymbol{\theta} - \gamma T \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} & \text{otherwise,} \end{cases}$$

where $\boldsymbol{g}$ is the gradient, γ is the learning rate, and T is the gradient threshold.

5 Autoencoders

Autoencoders are *generative models*. This means that their objective is to learn the underlying hidden structure of the data. They aim to model the distribution $p_{\text{model}}(\boldsymbol{x})$ that resembles $p_{\text{data}}(\boldsymbol{x})$ to generate new samples. Autoencoders are an *explicit* generative model, which means that they explicitly define the probability distribution $p_{\text{model}}(\boldsymbol{x})$ and then sample from it to generate new data points.

In machine learning, we often have high-dimensional data $\boldsymbol{x} \in \mathbb{R}^n$, such as images, audio, or time-series. Hence, it is crucial to find a low-dimensional representation that can effectively compress the data while preserving its essential information.

Autoencoders offer a solution by making use of the *encoder-decoder structure*; see Figure 14. The *encoder* f projects the input space $\mathcal{X}$ into a latent space $\mathcal{Z}$, while the *decoder* g maps the latent space $\mathcal{Z}$ back to the input space $\mathcal{X}$. The assumption made by the autoencoder architecture is that if the decoder is capable of reconstructing the original input solely from the compressed representation, then this compressed representation must be meaningful. Consequently, the composition $g \circ f$ aims to approximate the identity function on the data for a low reconstruction error.

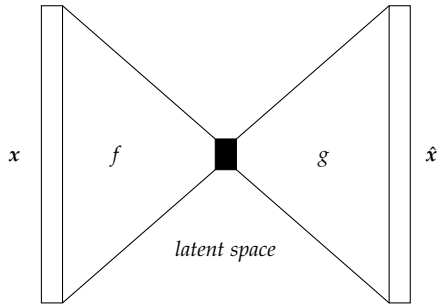

Figure 14. Autoencoder architecture.

Furthermore, to enable the generation of new samples from the latent space, the latent space must be well structured, characterized by *continuity* and *interpolation*. Continuity means that the entire space must be covered by the data points, while interpolation means that if we interpolate between two points, then the interpolation must also be a well behaved data point.

5.1 Linear autoencoders

If we restrict f and g to be linear, the encoder f becomes equivalent to the projection performed by *principal component analysis*. The advantage of such a reconstruction is that it can be found in a closed form. However, it is not very powerful.

5.2 Non-linear autoencoders

We can gain a lot of performance by allowing f and g to be non-linear. In this case, the encoder and decoder are implemented as neural networks. To train these networks, we optimize for the reconstruction error,

$$\boldsymbol{\phi}^\star, \boldsymbol{\psi}^\star \in \operatorname*{argmin}_{\boldsymbol{\phi},\boldsymbol{\psi}} \sum_{n=1}^{N} \|\boldsymbol{x}_n - g_{\boldsymbol{\psi}}(f_{\boldsymbol{\phi}}(\boldsymbol{x}_n))\|^2$$

We can distinguish between *undercomplete* and *overcomplete* latent spaces. A latent space is undercomplete if $\dim(\mathcal{Z}) < \dim(\mathcal{X})$, while it is over-

complete if $\dim(\mathcal{Z}) > \dim(\mathcal{X})$. The idea of an undercomplete hidden representation is to enable the network to learn the important features of the data by reducing the dimensionality of the hidden space. This prevents the autoencoder from simply copying the input and forces it to extract meaningful and discriminative features. An overcomplete latent spaces are useful for denoising and inpainting autoencoders, where we have an imperfect input and want a perfect output. The overcompleteness allows the model to extract more features from the transformed input, leading to improved performance.

5.3 Variational autoencoders

While autoencoders are good at reconstruction, they struggle at generating new high quality samples, which is due to the lack of continuity in the latent space. There are large regions in the latent space where there are no observations, thus the model does not know what to output when it get an input from those regions.

Variational autoencoders (VAE) are designed to have a continuous latent space. It achieves this by making the encoder output a probability distribution over latent vectors, rather than a single latent vector. Generally, it outputs a mean vector $\boldsymbol{\mu}$ and standard deviation vector $\boldsymbol{\sigma}$. The idea is that even for the same input, the latent vector can be different, but in the same area. This means that data points cover areas in the latent space, rather than single points, ensuring continuity.

However, since there are no limits on the values taken by $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, the encoder may learn to generate very different $\boldsymbol{\mu}$ for each class while minimizing $\boldsymbol{\sigma}$. This would mean that the encoder essentially outputs points again to decrease the reconstruction error. We can avoid this by minimizing the KL-divergence[10] between the output distribution and a standard normal distribution. Intuitively, this encourages the encoder to distribute the encodings evenly around the center of the latent space.

To train the model, we want to maximize the likelihood of the training data,

$$p(\boldsymbol{x}) = \int_{\boldsymbol{z}} p(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z}) \mathrm{d}\boldsymbol{z}.$$

However, this is intractable. Thus, we define an approximation of the posterior, $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$, which is computed by the encoder. We can now

[10] The KL-divergence is defined as

$$D_{\mathrm{KL}}(p \| q) \doteq \mathbb{E}\left[\log\left(\frac{p(x)}{q(x)}\right)\right].$$

It is not symmetric and non-negative.

derive the *evidence lower bound* (ELBO),

$$
\begin{aligned}
\log p(\boldsymbol{x}) &= \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}[\log p(\boldsymbol{x})] && \boldsymbol{x} \text{ does not depend on } z. \\
&= \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)p(z)}{p(z \mid \boldsymbol{x})}\right] && \text{Bayes' rule.} \\
&= \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}\left[\log\left(\frac{p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)p(z)}{p(z \mid \boldsymbol{x})}\frac{q_{\boldsymbol{\phi}}(z \mid \boldsymbol{x})}{q_{\boldsymbol{\phi}}(z \mid \boldsymbol{x})}\right)\right] && {}^{q(z|\boldsymbol{x})}/_{q(z|\boldsymbol{x})} = 1. \\
&= \mathbb{E}_{z|\boldsymbol{x}}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)] - \mathbb{E}_{z|\boldsymbol{x}}\left[\log \frac{q_{\boldsymbol{\phi}}(z \mid \boldsymbol{x})}{p(z)}\right] + \mathbb{E}_{z|\boldsymbol{x}}\left[\log \frac{q_{\boldsymbol{\phi}}(z \mid \boldsymbol{x})}{p(z \mid \boldsymbol{x})}\right] \\
&= \mathbb{E}_{z|\boldsymbol{x}}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)] - D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(z \mid \boldsymbol{x})\|p(z)) + D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(z \mid \boldsymbol{x})\|p(z \mid \boldsymbol{x})) \\
&\geq \mathbb{E}_{z|\boldsymbol{x}}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)] - D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(z \mid \boldsymbol{x})\|p(z)). && \text{KL-divergence is non-negative.}
\end{aligned}
$$

The first term of the ELBO encourages low reconstruction error, while the second term makes sure that the approximate posterior $q_{\boldsymbol{\phi}}$ does not deviate too far from the prior p.

A minor problem is that, during training, we cannot compute the derivative of expectations w.r.t. the parameters that we wish to optimize. Thus, we must use the *reparametrization trick*, which involves treating the random sampling as a single noise term. In particular, instead of sampling $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}))$, we sample $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and compute $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$. Using this trick, we can remove the mean and variance from the sampling operation, meaning that we can differentiate w.r.t. the model parameters.

5.4 β-VAE

VAEs still have problems with their latent space; the representations are still *entangled*. This means that we do not have an explicit way of controlling the output. For example, in the MNIST dataset, we have no way of explicitly sampling a specific number. The β-VAE solves this problem by giving more weight to the KL term with an adjustable hyperparameter β that balances latent channel capacity and independence constraints with reconstruction accuracy. The intuition behind this is that if factors are in practice independent from each other, the model should benefit from disentangling them.

In practice, we want to force the KL loss to be under a threshold δ,

$$
\begin{aligned}
\underset{\boldsymbol{\phi},\boldsymbol{\psi}}{\text{maximize}} \quad & \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}}\left[\mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}\left[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)\right]\right] \\
\text{subject to} \quad & D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(z \mid \boldsymbol{x})\|p(z)) \leq \delta.
\end{aligned}
$$

Rewriting this as a Lagrangian, we get

$$
\begin{aligned}
\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\psi}, \beta) &= \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)] - \beta(D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}\|p(z)) - \delta) \\
&= \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)] - \beta D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}\|p(z)) + \beta\delta \\
&\geq \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid z)] - \beta D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}\|p(z)).
\end{aligned}
$$

Thus, this becomes our new objective function that we wish to maximize.

References

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.