Introduction to Machine Learning (for AI)

Neural Networks

Dr. Matias Valdenegro

December 12, 2024

Today's Lecture

In this lecture we (finally) cover neural networks. This is probably one of the reasons you are here.

We start with the perceptron, from there build into multi-layer perceptrons, into convolutional and recurrent neural networks.

We do mention backpropagation but we do not cover it in detail, as modern neural networks are trained using automatic differentiation.

Outline

What are Neural Networks?

Most if not all models we have seen previously are linear models[1]. Neural networks are non-linear models that are built by composing **layers**, which perform some computation and can have learnable weights, allowing for much more complex and non-linear behavior.

Many models we have covered previously are particular cases of neural networks, as a neural network is a more general concept that covers linear and non-linear models.

[1] Exception of decision trees and Kernel SVMs

Perceptron as Linear Binary Classifier

The basic model of a Perceptron is:

$$f(x) = \text{sgn}\left(\sum_i w_i x_i + b\right) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + b) \tag{1}$$

This model is called perceptron. The function sgn is the sign function defined by:

$$\text{sgn}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{2}$$

Also the step function can be used:

$$\text{step}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{3}$$

Perceptron Learning Rule

To learn the weights, this simple rule is applied iteratively:

1. Initialize all weights w_i with random values in a small range (like $[-0.5, 0.5]$).
2. For each data point (x_i, y_i) in the training set.
   2.1 Compute current output $\hat{y}_j = \text{sgn}(\mathbf{x} \cdot \mathbf{w} + b)$
   2.2 Update each weight w_k:

$$w_k(n+1) = w_k(n) + \alpha(\hat{y}_i - y_i)x_{ik} \tag{4}$$

3. Repeat step 1 up to n times, where n is the number of total iterations.

This assumes a representation where the bias is integrated into the weights (and a column of ones is added to the input features), and α is a learning rate.

Geometric Interpretation

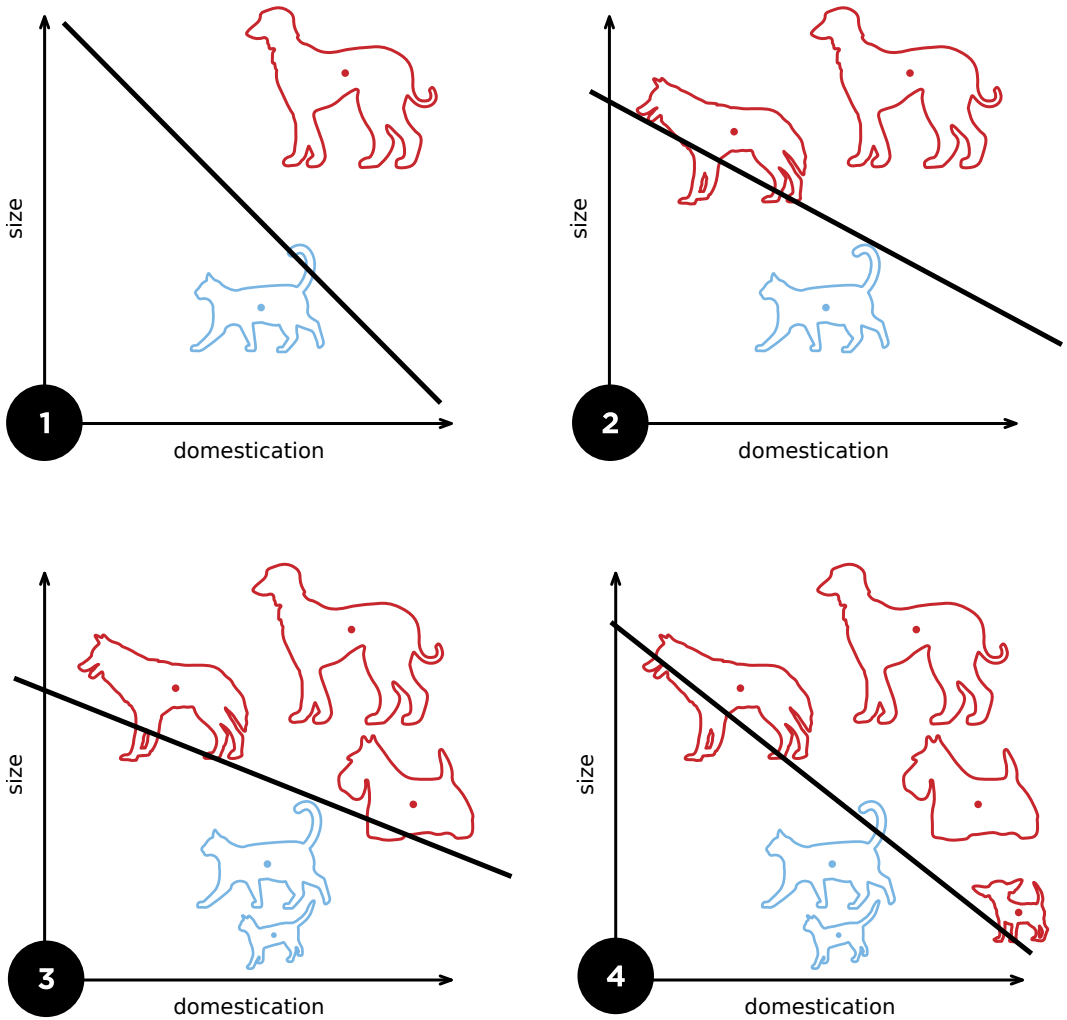


Figure from `https://en.wikipedia.org/wiki/Perceptron#/media/File:Perceptron_example.svg`

Multi-Layer Perceptrons

The Perceptron is a linear classifier, and if used for regression, it is the same as linear regression, without the step/sign function.
You can think the perceptron as being one layer, that computes:

$$f(x) = \sigma(\mathbf{w} \cdot \mathbf{x} + b) \tag{5}$$

Where now σ is an activation function, meant to introduce non-linear behavior into the model. This is of course different than Kernels in SVMs.

Multi-Layer Perceptrons

To increase the flexibility and capabilities of the model even further, one can then compose a sequence of L layers

$$\begin{aligned}
\mathbf{h}_0 &= \mathbf{x} \\
\mathbf{h}_1 &= \sigma(\mathbf{W}_1^\intercal \mathbf{h}_0 + \mathbf{b}_1) \\
&\dots \\
\mathbf{h}_L &= \sigma(\mathbf{W}_L^\intercal \mathbf{h}_{L-1} + \mathbf{b}_L)
\end{aligned}$$

Which we call the **Multilayer Perceptron** or Feedforward Neural Network. Note that each layer has its own set of weights and biases.

MLP Terminology

Layer The basic unit of computation (in a NN) that includes weights, biases, and a (non-linear) function. They are the basic building block of a neural network.

Activations The values $\mathbf{h}$ are called activations, and they are the outputs of a layer given some input $\mathbf{x}$.

Activation Functions σ is called an activation function, meant to introduce non-linearity into the output of the model.

Neurons

The most basic layer of a neural network is the Perceptron layer, also known as Linear (in PyTorch) or Dense (in Keras/TensorFlow). It computes $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$.

A single perceptron computes a scalar value $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$.

This is the computation that a single neuron makes, the general case is a layer with M neurons, each having its own set of weights $\mathbf{W}$ and biases b.

Neurons

A perceptron layer (Dense layer), contains M neurons, then its computation can be generalized to matrix form:

$$\mathbf{f}(x) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{6}$$

Where now $\mathbf{W}$ is a $M \times D$ matrix and $\mathbf{b}$ is a $M \times 1$ vector.

For the general case, the perceptron is not a classifier but a general feature learning layer. It inputs a vector $\mathbf{x}$, and outputs another vector $\mathbf{f}(x)$.

Dense Layer Parameters

A dense layer has some hyper-parameters that need to be set by the designer:

Number of Neurons Defines the number of neurons, has relationships with feature learning and task complexity. Hyper-parameter to be tuned (with grid search for example).

Activation Function Defines the non-linear behavior of the network, there are many many activation functions.

Tasks with MLPs

The last layer in your neural network usually defines the task to be solved.

Classification Use a Dense layer with $M = C$ neurons, and a softmax activation, with a cross-entropy loss.

Regression Use a Dense layer with M neurons (same dimensionality as your target value), a linear or sigmoid activation, and a mean squared error loss.

For regression, if the target is normalized to $[0, 1]$ then a sigmoid activation is preferable. If the range is not normalized, then use a linear activation (equivalent to no activation).

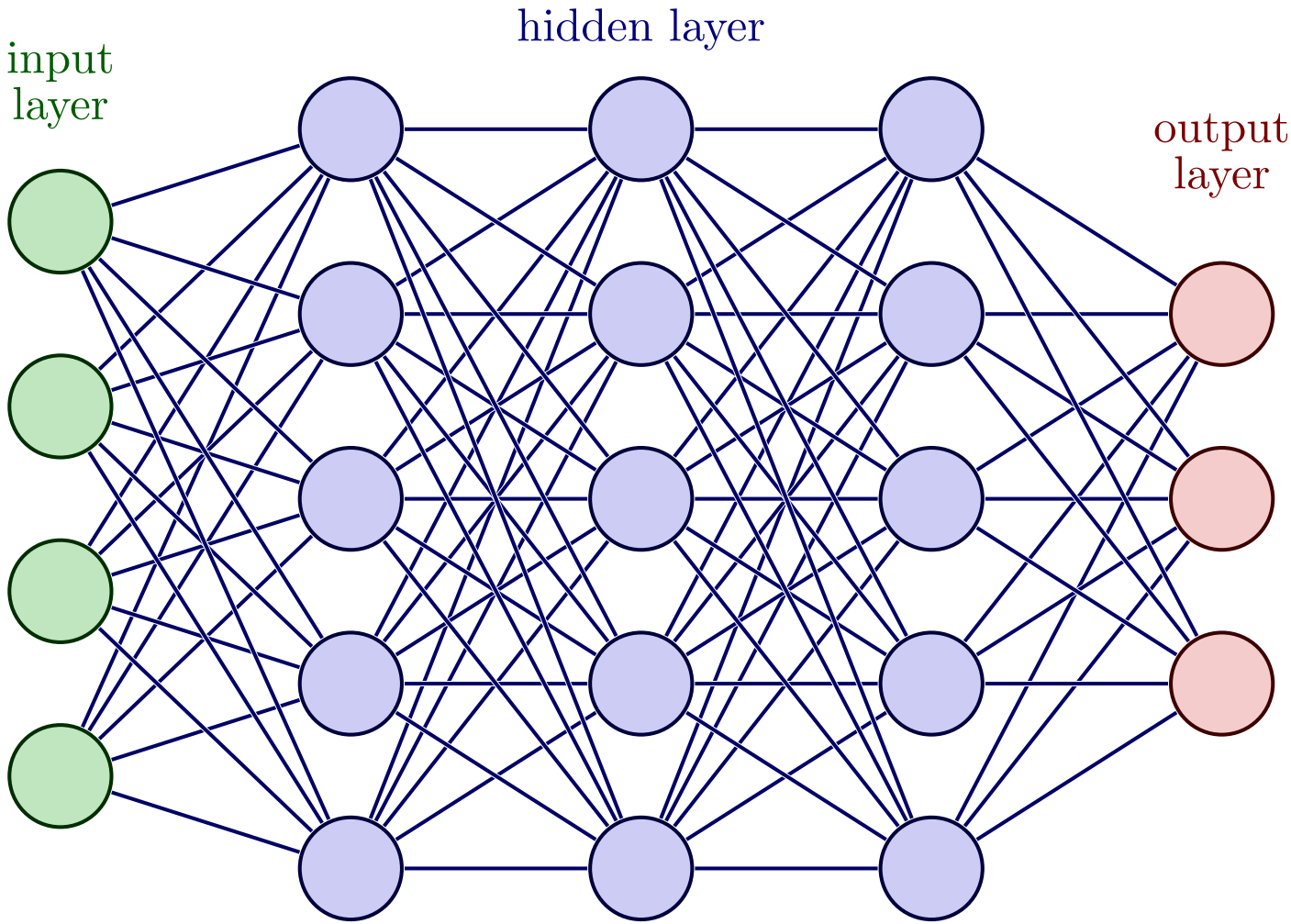
Number of Parameters

A Dense layer has total number of parameters $M \times (D + 1)$, as the weight matrix is $M \times D$ and the bias has M elements.

Each Dense layer adds parameters to a neural network, increasing the amount of non-linearity in the model outputs.

The number of total parameters P is an important parameter to be controlled, as too large leads to overfitting, and too small leads to underfitting. A Neural Network with its many variants is the first model (in this course) where you can tune P by making changes to the model (add or remove layers, make it deeper or shallower).

MLP Architecture/Structure



Learning Process

Now the important question is, how do we train MLPs? We use gradient descent and its variants. Considering $\mathbf{w} = [W_1, b_1, W_2, b_2, ..., W_L, b_L]$:

$$\mathbf{w}_{m+1} = \mathbf{w}_m - \alpha \frac{\partial \ell}{\partial \mathbf{w}} \tag{7}$$

Where α is the learning rate, which controls the step size in the weight space.

Gradient descent as a framework allows you to minimize any loss function, as long as it is differentiable and the gradient of the loss with respect to parameters $\mathbf{w}$ can be computed.

Stochastic Gradient Descent

So far we have covered gradient descent, which requires to compute the loss over all training samples:

$$\frac{\partial \ell}{\partial \mathbf{w}} = \sum_{i}^{N} \frac{\partial \ell(f(x_i), y_i)}{\partial \mathbf{w}} \tag{8}$$

With f being the forward pass of your neural network. Computing loss over all training samples is very cumbersome, an alternative is **Stochastic Gradient Descent**, where a non-overlapping batch/sample of the training set is used at a time:

$$\frac{\partial \ell}{\partial \mathbf{w}} = \sum_{i \in \text{batch}} \frac{\partial \ell(f(x_i), y_i)}{\partial \mathbf{w}} \tag{9}$$

This iterates over the training set in batches of size B.

Stochastic Gradient Descent

SGD itroduces noise into the learning process but it is usually tolerable. The loss and gradient is only computed in a subset of data at a time, called a batch. The length of this subset is called **batch size** (B).

With $B = 1$ it is called Stochastic Gradient Descent (SGD). For $B > 1$ it is called Mini-Batch Gradient Descent (MGD). But generally the literature and community overlooks these differences and calls SGD always.

Batch size B has to be tuned to memory/compute availability. It controls the quality/noise of the gradient. A whole pass over the a complete dataset is called an **epoch**.

Backpropagation

Since a MLP or any neural network is a composition of many non-linear functions $h_1, h_2, ..., h_L$, the gradient can be computed using the chain rule (from mathematics). The algorithm known as backpropagation is just an application of the chain rule.

$$\begin{aligned}
\mathbf{h}_0 &= \mathbf{x} \\
\mathbf{h}_1 &= \sigma(\mathbf{W}_1^\intercal \mathbf{h}_0 + \mathbf{b}_1) \\
&... \\
\mathbf{h}_L &= \sigma(\mathbf{W}_L^\intercal \mathbf{h}_{L-1} + \mathbf{b}_L) \\
\ell &= \text{loss}(\mathbf{h}_L, y)
\end{aligned}$$

Chain Rule for $y = f(g(x))$:

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} \tag{10}$$

$$\frac{\partial \ell}{\partial \mathbf{w}} = \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L_1}}\frac{\partial \mathbf{h}_{L_1}}{\partial \mathbf{h}_{L_2}} \cdots \frac{\partial \mathbf{h}_1}{\partial \mathbf{w}}$$

Backpropagation

Now for example, considering that $\mathbf{h}_2 = \sigma(\mathbf{u}_2)$ and $\mathbf{u}_2 = \mathbf{W}_2^\intercal \mathbf{h}_1 + \mathbf{b}_2$, then:

$$\begin{aligned}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} &= \frac{\partial \mathbf{h}_2}{\partial \mathbf{u}_2}\frac{\partial \mathbf{u}_2}{\partial \mathbf{h}_1} \\ &= \frac{\partial \sigma(\mathbf{u}_2)}{\partial \mathbf{u}_2}\frac{\partial \mathbf{W}_2\mathbf{h}_1}{\partial \mathbf{h}_1}\end{aligned}$$

The term $\frac{\partial \sigma(\mathrm{u}_2)}{\partial \mathrm{u}_2}$ is the gradient of the activation function, while $\frac{\partial \mathrm{W}_2\mathrm{h}_1}{\partial \mathrm{h}_1} = \mathbf{W}_2$.

Similar derivations can be done for other layers we will cover later in this lecture, but we will use another more easy framework.

Terminology

Forward Pass

This is computation of the neural network output given an input, that is applying each layer from the input to the last layer.

Backward Pass

Computing the gradient of the loss function given an input $\frac{\partial \ell}{\partial \mathrm{w}}$, and going backwards through the layers and compute local gradients $\frac{\partial \mathrm{h}_L}{\partial \mathrm{h}_{\mathrm{L}_1}}$, in order to compute the gradient of the loss.

Automatic Differentation - Autograd

Computing gradients in analytical forms (by hand) is of course cumbersome, modern computational libraries can compute gradients automatically. Two possible ways.

Forward Mode Traverses the chain rule from inside to outside (tensorflow graph mode).

Reverse Mode Traverses the chain rule from outside to inside (pytorch, tensorflow eager mode).

Forward Mode usually requires to build a computational graph representation and could be more optimal, but reverse mode is straightforward. Each operation needs to have gradient operations implemented.

Weight Initialization and Symmetry

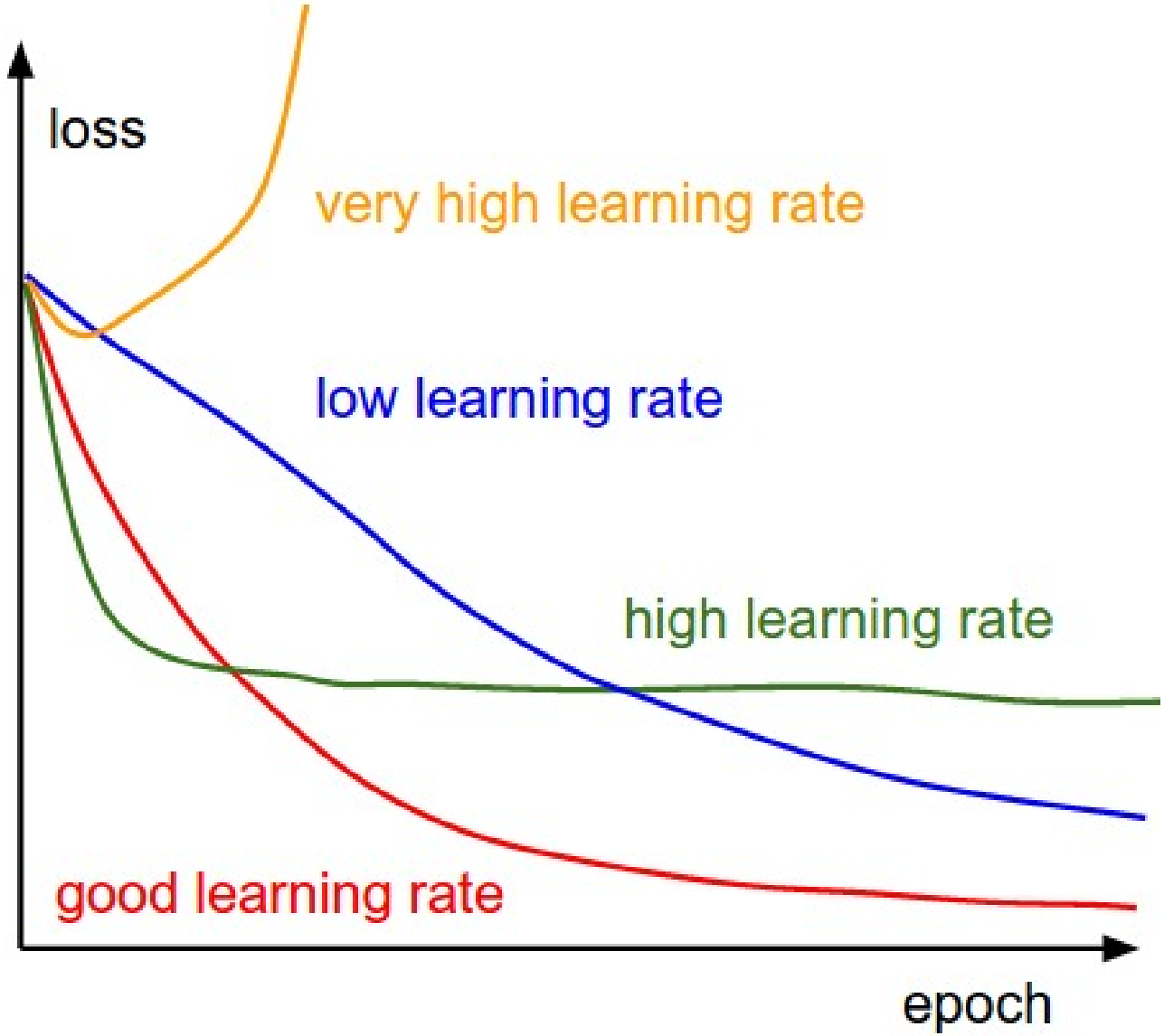
Weights need to have a starting value to use gradient descent, They are usually initialized to a small random range, using a Gaussian or Uniform distribution in range $[-0.5, 0.5]$. There are many methods for particular initialization of weights.

One important principle is that if all weights are set to zero, then the gradients are exactly zero always, and training cannot proceed. Initializing weights randomly breaks this, which is called breaking the symmetry.

Tuning the Learning Rate

- The most important parameter to have successful training is the learning rate, and the number of training epochs.
- The idea is to set the learning rate to a value that produces consistent decrease of the loss value across batches.
- The number of epochs should be set to a value makes sure that the loss has converged to a small value. Not enough epochs will produce underfitting.

Tuning the Learning Rate



Tuning the Learning Rate - Problems

- **High Learning Rate**. The loss diverges to infinity or NaN (not a number). In some cases the loss does not decrease instead, and oscillates.
- **Low Learning Rate**. The loss decreases but slowly, making training unnecessarily long.
- **Good Learning Rate**. The loss decreases quickly, converging into a reasonable number of epochs.

Tuning the Learning Rate - Solutions

- **High Learning Rate**. Divide the current learning rate by 10, until the loss decreases consistently.
- **Low Learning Rate**. Carefully increase the learning rate, by factors of 2-5.
- **Good Learning Rate**. Keep this learning rate.

Activation Functions

Name	Range	Function
Linear	$[-\infty, \infty]$	$g(x) = x$
Sigmoid	$[0, 1]$	$g(x) = (1 + e^{-x})^{-1}$
Hyperbolic Tangent	$[-1, 1]$	$g(x) = (e^{2x} - 1)(e^{-2x} + 1)^{-1}$
ReLU	$[0, \infty]$	$g(x) = \mathsf{max}(0, x)$
Softplus	$[0, \infty]$	$g(x) = \ln(1 + e^{x})$
Softmax	$[0, 1]^n$	$g(\mathbf{x}) = (e^{x_i})(\sum_k e^{x_k})^{-1}$
LeakyReLU	$[-\infty, \infty]$	$g(x) = \mathsf{max}(0.01x, x)$
Parametric ReLU	$[0, \infty]$	$g(x) = \mathsf{max}(\alpha x, x)$
Swish	$[-1, \infty]$	$g(x) = x \cdot \mathrm{sigmoid}(x)$

ReLU is a good default for hidden layers, output layers use a task-dependent activation, softmax for classification, and linear, softplus, or sigmoid for regression.

Issues with Training

Training a neural network is not an easy task compared with linear models we covered before, here are common issues.

Vanishing Gradients. Using saturating activations (sigmoid and tanh) have close to zero gradient at the extremes of their inputs, which produces tiny gradients while training NNs using this activation in the hidden layers. A solution is to use non-saturating activations like the ReLU.

Non-Convex Loss. The loss function landscape is not convex due to the neural network being non-linear, which introduces issues during optimization, like flat regions, saddle points, and multiple local and global optima.

Issues with Training

Potential Overfitting. Since the number of parameters P is now tunable, depending on number of layers and number of neurons, then there is a large potential for overfitting. This can be solved with careful network design, and regularization.

Hyper-Parameters. Neural networks have more hyper-parameters that need setting (number of layers, number of neurons), and even more in the training process (batch size, learning rate). A lot of knobs need to be moved correctly for things to work.

Outline

Using MLPs with Image Inputs

- A M-neuron layer connected to an $W \times H$ image will have $M \times W \times H$ weights.
- Too many weights to be learned.
- No translation equi-invariance is built into the network design.
- In summary, this is far from ideal.

Convolutional Neural Networks (CNNs)

- CNNs impose constraints on the weights in order to account for a more specialized input data **?**.
- CNNs replace the matrix multiplication in fully connected ANNs for the convolution operator.

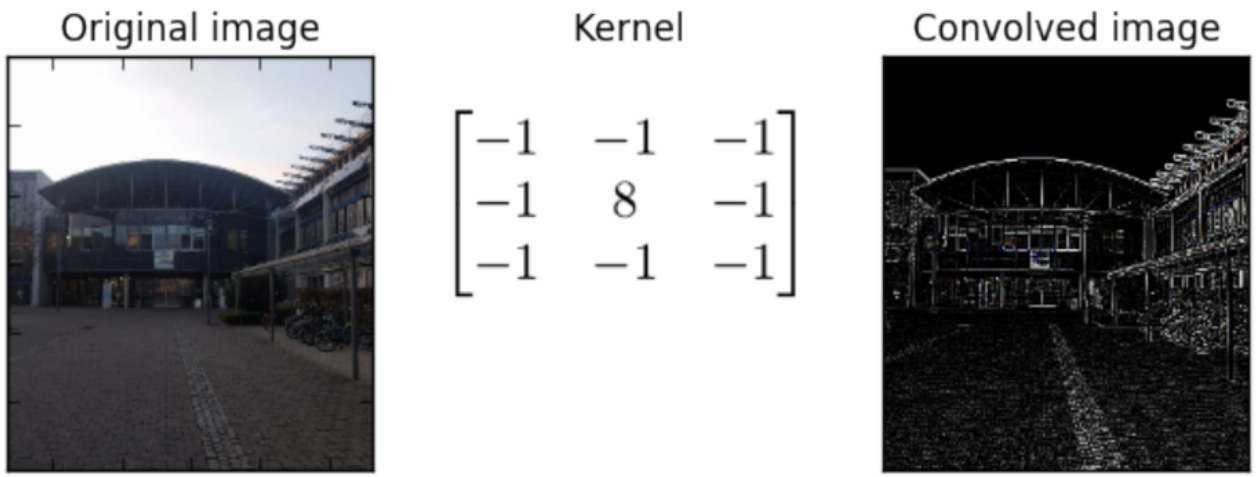


Figure: Convolution operation for image processing.

Model Priors in CNNs

- Local connectivity
- Weight sharing

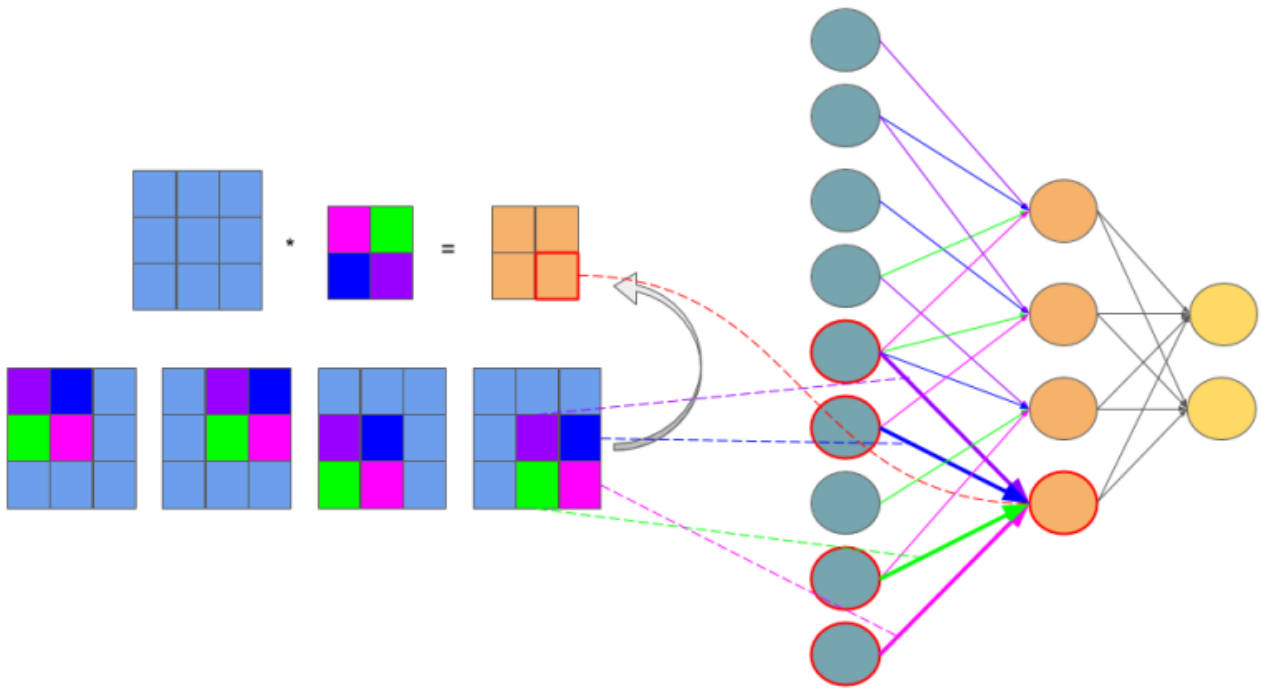


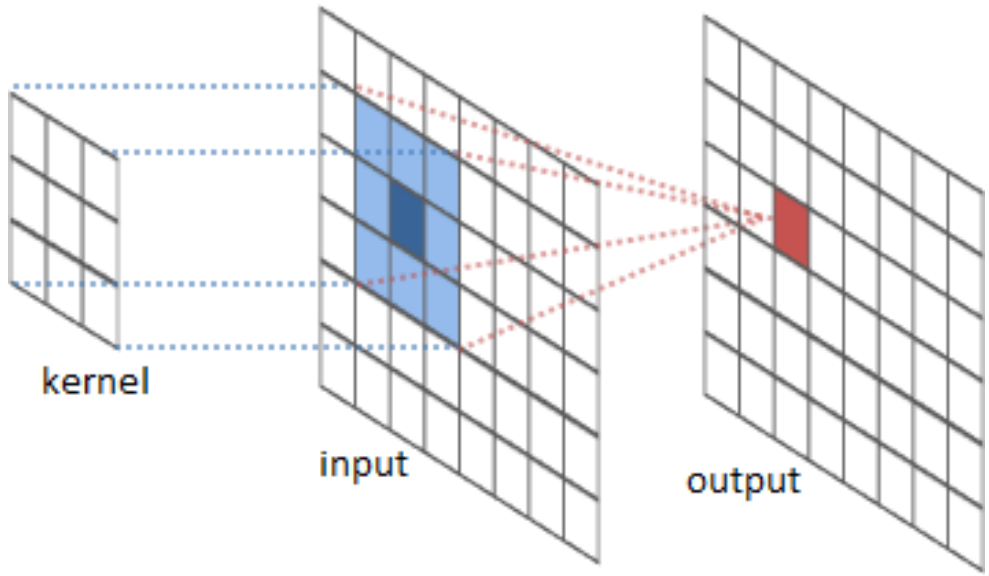
Figure: **?**

Convolutional Neural Networks

In the 80's Yann LeCun had the following idea:

- Connect a neuron only to a spatial neighborhood of the image and slide it on the image.
- This learns the same weights independent of location in the image. **Less weights to learn**.
- This is naturally represented as convolution, where the convolution filter contains the neuron weights.
- To reduce the amount of information, subsample the outputs after convolution.

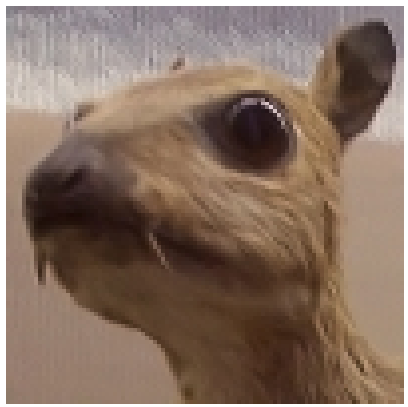
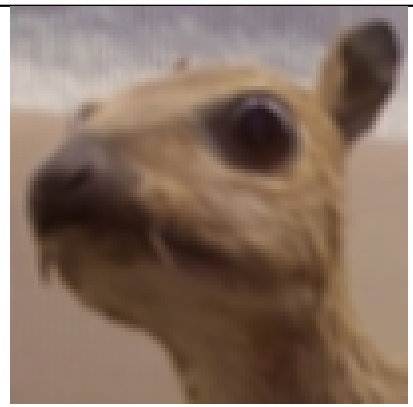
Convolution Operation



$$\text{out}(x, y) = \sum_i \sum_j \text{input}(x + i, y + j)\, \text{kernel}(i, j)$$

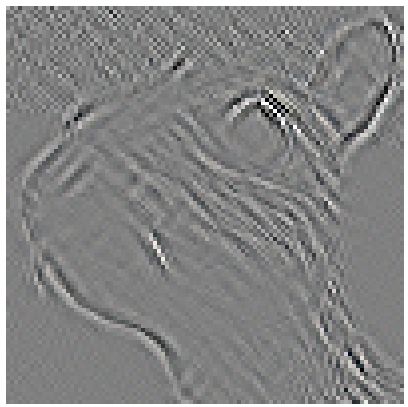
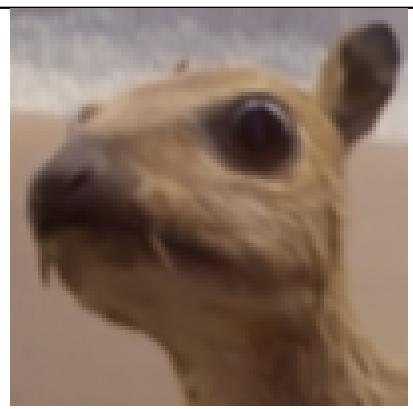
The i and j are pixel coordinates over the kernel dimensions (width and height), while x and y are pixel coordinates over the image/feature map dimensions.

Convolution Kernels / Filters

Name	Kernel Matrix	Output Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Box Blur	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

Images taken from `https://en.wikipedia.org/wiki/Kernel_(image_processing)`

Convolution Kernels / Filters

Name	Kernel Matrix	Output Image
Laplacian	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Gaussian Blur	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Images taken from `https://en.wikipedia.org/wiki/Kernel_(image_processing)`

Idea of Convolutional Networks

- Use learnable filter kernels (parameters or weights).
- Build multiple 2D **feature maps** (output of convolution).
- Each feature map in one layer is connected to each feature map from the previous layer and ...
- we use different filter kernels for each pair of feature maps.
- **We exploit the structure of images: in comparison to a fully connected layer, we make use of the spatial order of the pixels**

$\rightarrow$ We reduce the number of connections significantly and the number of weights even more.

$\rightarrow$ We gain a little bit rotation and illumination invariance.

Convolutional Layers

Purpose: Dimensionality reduction, feature learning, weight sharing
Forward pass:

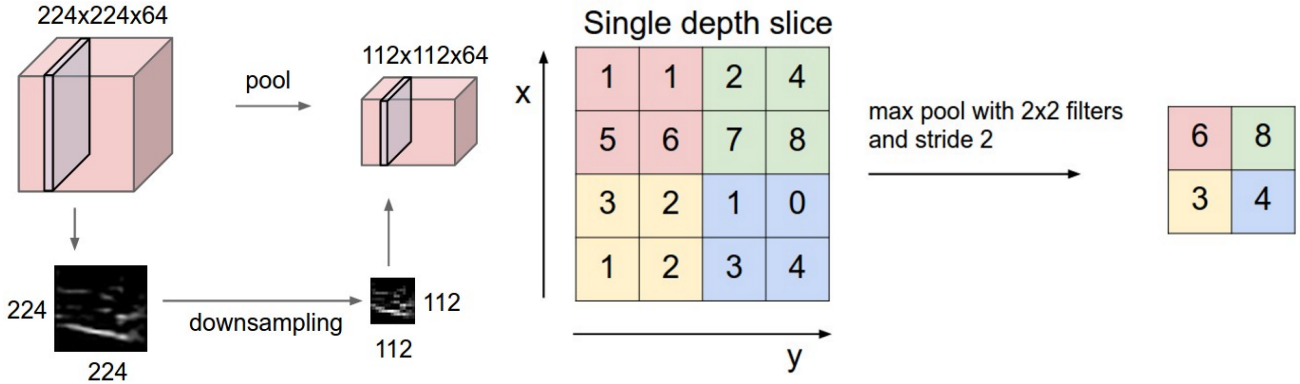
$$\mathbf{Z}^j = \sum_{i=0}^{I-1} \mathbf{W}^{ji} * \mathbf{X}^i + b^{ji}, \quad \mathbf{Y}^j = g(\mathbf{Z}^j),$$

where

- $\mathbf{X}^i$ is the i-th feature map in the input
- $\mathbf{Y}^j$ is the j-th feature map in the output
- $\mathbf{W}^{ji}$ is the filter kernel and b^{ji} the bias between feature maps i and j
- Note that the filter kernel will usually be moved by only one pixel in each step (they have a stride of 1), i.e. they overlap.

Pooling Layers

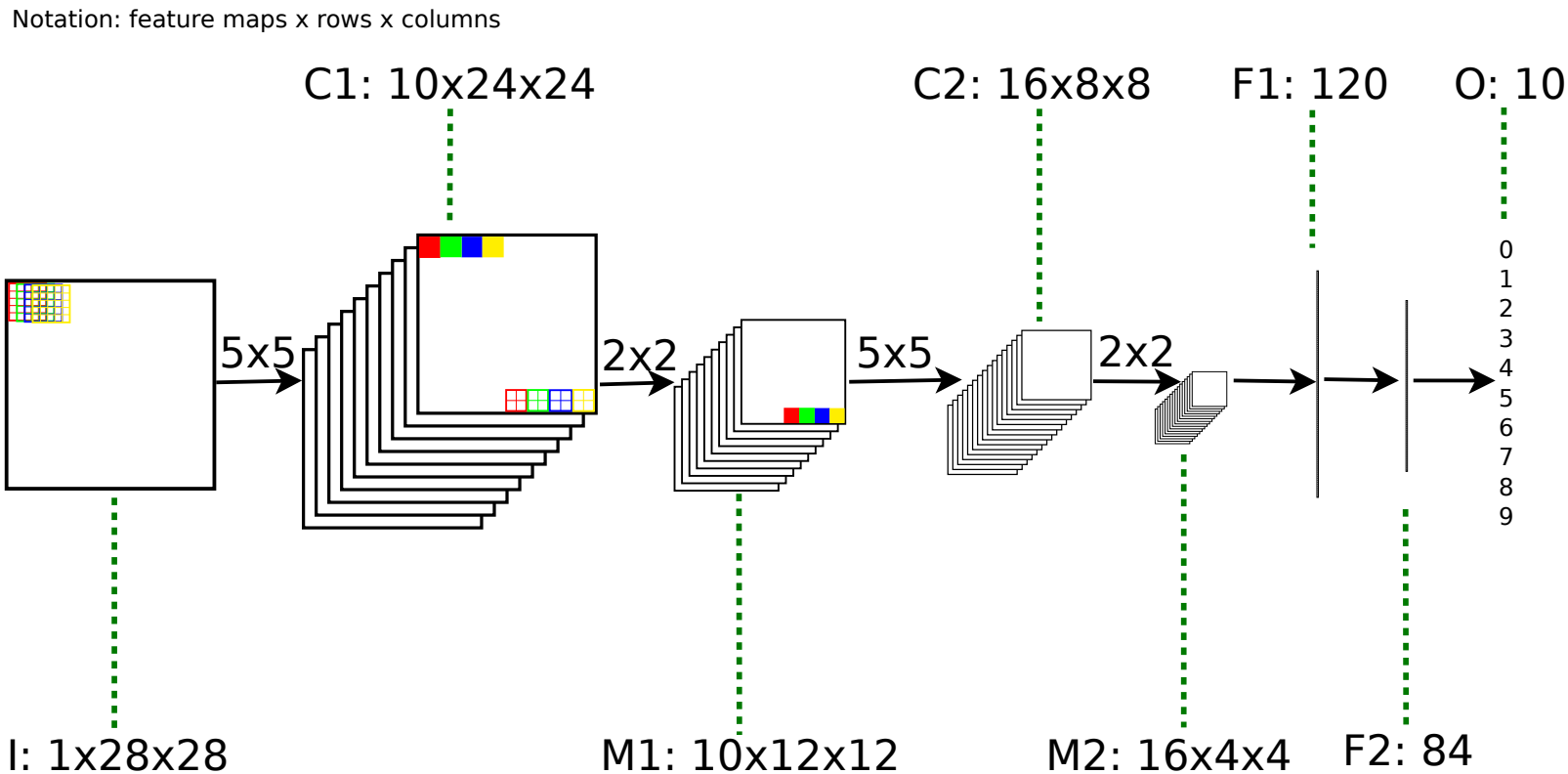
Purpose: dimensionality reduction and local translation invariance



Max-pooling example (source: `http://cs231n.github.io/`)

- Summarize low-level features in the input region, and reduce feature dimensions and number of parameters
- Pooling function can be max or average.
- Global (max/average) pooling performs a single operation on the whole feature map, producing a 1×1 spatial output.

Convolution / Pooling



Building Convolutional Networks

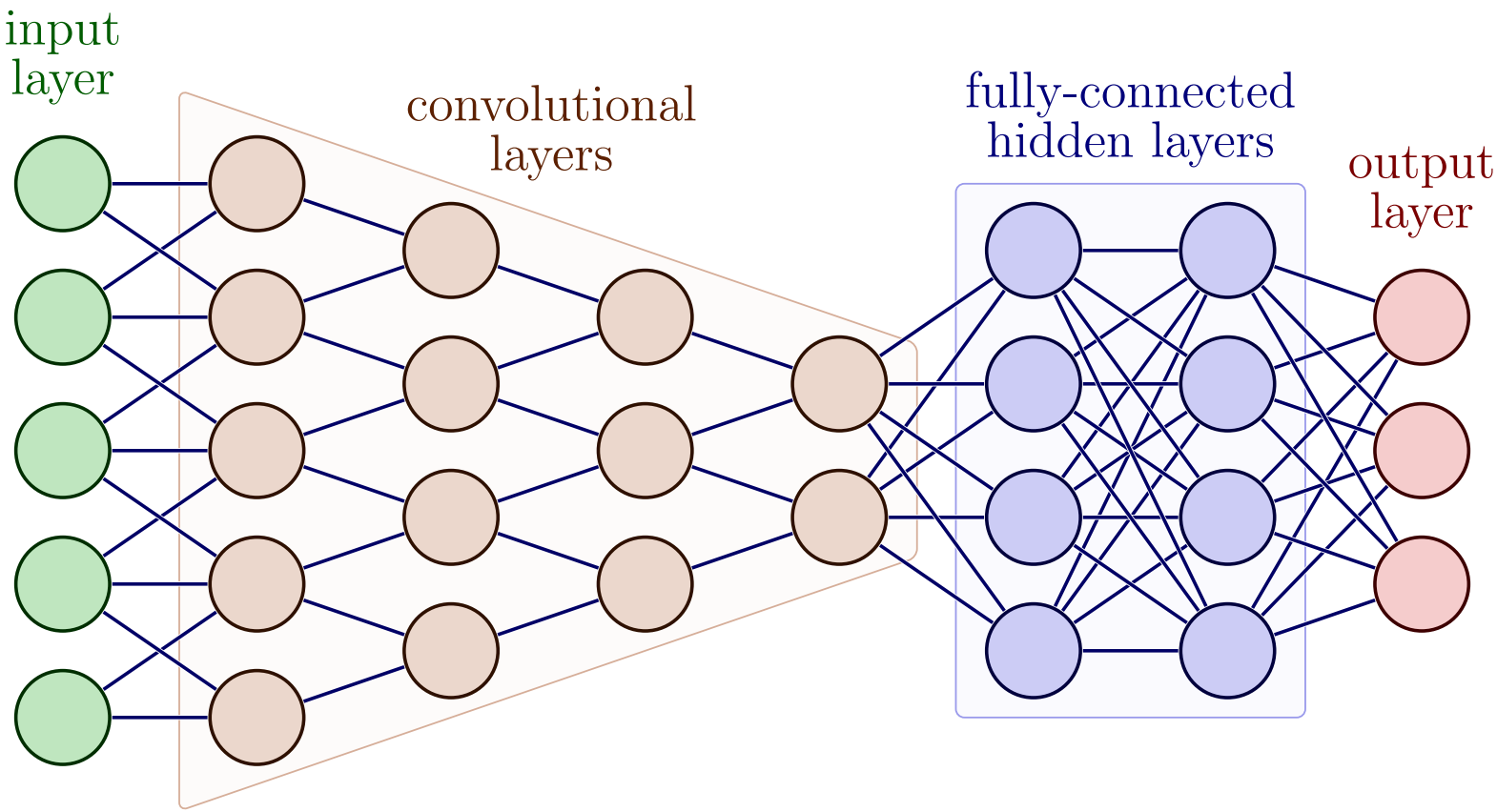
To build a convolutional neural network, convolutional and max-pooling layers are used.

Feature Extraction Stacks of convolution and max-pooling layers are used to extract hierarchical features, finishing with flattening to produce a large feature vector.

Task (Classification or Regression) A stack of two or more Dense layers are fed with the feature vector from the previous set of layers, and learn to perform the final task.

The whole CNN is trained using stochastic gradient descent, so weights for feature extraction (convolutional layers) and task (dense layers) are learned end-to-end.

Convolutional NNs



Feature Hiearchy in CNNs

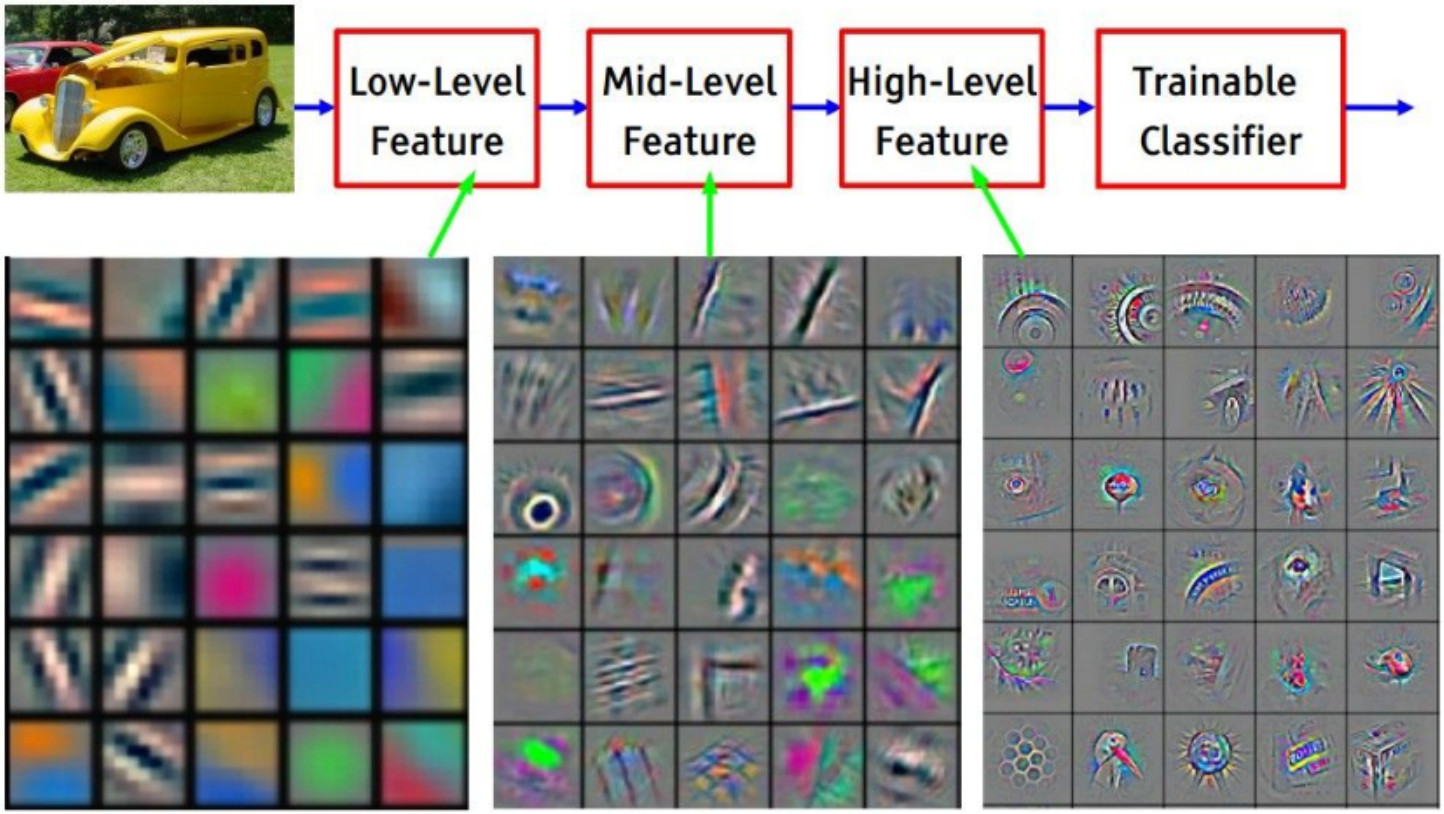


Figure: Convolutional features. xA CNN learns that images consists of edges, then groups edges into parts, and groups them into more complex objects. This maps directly from the feature extraction part of a CNN.

Input-Output Shapes

Layer	Input Shape	Output Shape	Remarks
Fully Connected	$(f,)$	$(n,)$	Only operates on last dimension of input
Convolutional	(w_i, h_i, c_i)	(w_o, h_o, c_o)	$x_o = \frac{x_i - f + 2p + 1}{s}$
Pooling	(w_i, h_i, c)	$(\frac{w_i}{p}, \frac{h_i}{p}, c)$	p is pooling factor (usually $p = 2$)
Global Pooling	(w_i, h_i, c)	$(1, 1, c)$	Consumes spatial dimensions

Notation. f for input feature dimensionality or filter size, c for channels, n for number of neurons, p for padding, and s for strides. Convolutional dimension can be maintained if you set $p = \frac{f-1}{2}$ with $s = 1$.

Outline

Recurrent Neural Networks

An RNN is a neural network that has three major differences over a feed forward network:

1. An RNN has memory, through an internal state that is persisted between timesteps.
2. The internal state (denoted by h) is meant to capture temporal or sequence information, which is also learned from data.
3. An RNN is able to receive variable-sized inputs, and to produce variable-size outputs, usually with an encoder-decoder architecture.

Recurrent Neural Networks

In an RNN, h_t is the hidden state for timestep t, x_t is the input given at timestep t, and y_t is the output produced at each timestep.

$$\mathbf{h}_t = \sigma_h(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \tag{11}$$

$$\mathbf{y}_t = \sigma_y(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y) \tag{12}$$

These equations are vectorized if features are multi-dimensional. Note that instead of a single weight/bias matrix, there are three weight matrices $(\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy})$ and two bias vectors $(\mathbf{b}_h, \mathbf{b}_y)$. σ denotes activation functions, one for hidden state and another for the output.

Notation

Symbol	Meaning
t	Timesteps, temporal dimension.
x	Input values.
y	Output values.
h	Hidden state.
W	Weigh matrices,
b	Bias vectors.
$\cdot$	component-wise product.

Recurrent Neural Networks

- An important feature of an RNN is that the weight/bias matrices are shared among timesteps.
- This means that these weights/biases are the same across timesteps, the operation that they represent is invariant to the sequence.
- This has the advantage of reducing the number of parameters needed to learn a task, and to force the RNN cell to learn timestep-invariant features. It is similar to how CNNs work with respect to translation invariance.

Recurrent Neural Networks

- The dimensionality of the hidden state h_t does not have to be the same as the input.
- As in feedforward NNs, an RNN could project the input into a higher dimensional space, which also contains the hidden state.
- This allows for sequence features in the hidden state that activate or correlate with features in the sequence, modeling long and short term dependencies.

RNN Hyper-Parameters

- The most basic hyper-parameter is the number of units/neurons in a recurrent layer, which also defines the hidden space dimensionality.
- Activations σ_h is usually sigmoid or tanh, while activation σ_y can be configurable by the user. The selection of σ_h is done in a way to prevent the hidden state from growing without control.
- An RNN can be configured to return a sequence of outputs y_t, or only the output of the last timestep y_n.
- RNNs can also be stateful, meaning that the hidden state is kept across batches of inputs, and the state can be manually reset (to zero).

Fundamental Issues in RNNs

Vanishing and Exploding Gradients. RNNs are prone to have vanishing or exploding gradients, due to the recurrent application of several weight matrices, from where gradient increases or decreases without control. Much more likely to happen for long sequences.

Long-Term Dependencies. RNNs have trouble correctly modeling dependencies across large sequences, as the hidden state (a vector) $\mathbf{h}_t$ has to encode information from across the sequence, and is effectively the only information shared across the sequence dimension.

Vanishing or Exploding Gradients

An RNN basically applies a repetitive multiplication by a weight matrix W. If we use an eigen decomposition $W = Q\Lambda Q^T$, then an approximation of the state h_t is:

$$h_t = Q^T \Lambda^t Q \tag{13}$$

Any eigenvalues inside Λ that are less than one will eventually converge to zero, and any eigenvalue bigger than one will explode into infinity. This happens when sequences are very long.

This then translates into vanishing and exploding gradients when training RNNs. This is because of the recurrent application of the kernel matrix.

Tricks for Exploding Gradients

Element-Wise Clipping

Given the gradient $\mathbf{g}$ of a batch, clip any component of the gradient that has absolute value bigger than v.

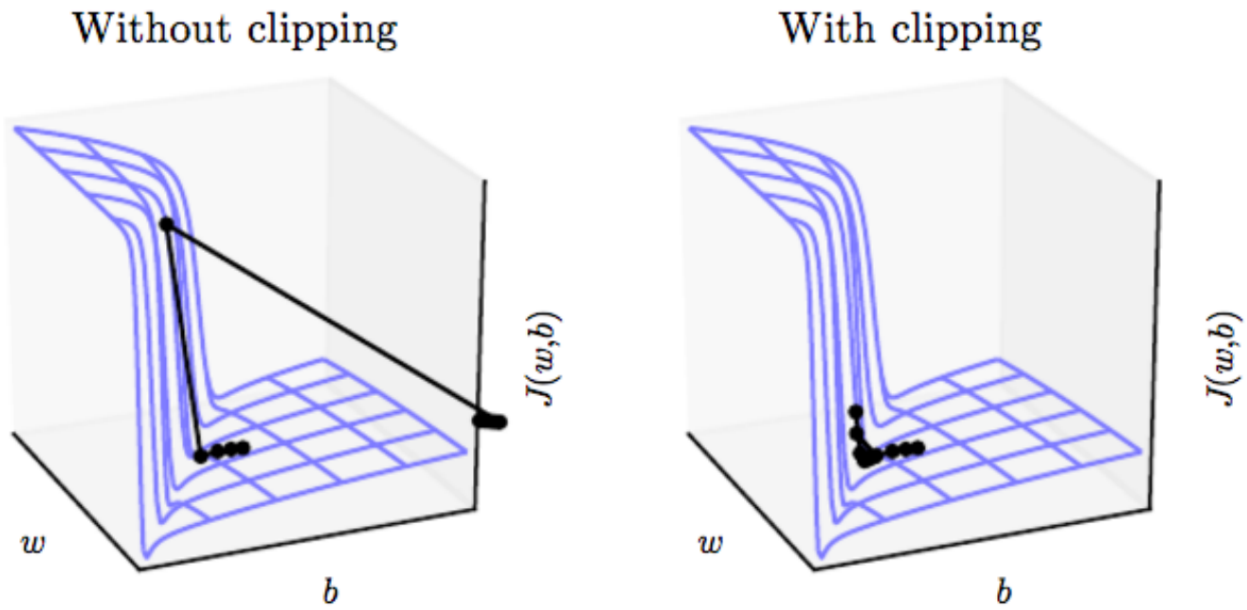
That is for element where $g_i > v$, set $g_i = v$.

Norm Clipping

Clip the norm of the gradient. If $||\mathbf{g}|| > v$ set $g = \frac{gv}{||g||}$.

The clip parameter v has to be decided. It can be obtained by monitoring gradients during training and trial and error.

Gradient Clipping [Goodfellow et al., 2016]



— Goodfellow et al., *Deep Learning*

This is a visual representation of exploding gradients, and how clipping can "solve" this problem. It is better to slowly go down the cliff than overshoot and go down the cliff fast.

Long-Short Term Memory [Hochreiter and Schmidhuber, 1997]

- Developed by Sepp Hochreiter and Jürgen Schmidhuber in 1997.
- An LSTM is a more advanced kind of RNN cell, which has a much smaller chance of producing vanishing or exploding gradients.
- Additionally, it has a more advanced ability to model long-term dependencies inside the sequence.
- It works by using gated connections, and by splitting the state h_t into parts that are useful for output prediction, and parts to learn features from the sequence.

Long-Short Term Memory

First, an LSTM computes three gates (forget $\mathbf{g}_f$, input $\mathbf{g}_i$, output $\mathbf{g}_o$):

$$\mathbf{g}_f = \sigma(\mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{xf}\mathbf{x}_t + \mathbf{b}_f) \tag{14}$$

$$\mathbf{g}_i = \sigma(\mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{W}_{xi}\mathbf{x}_t + \mathbf{b}_i) \tag{15}$$

$$\mathbf{g}_o = \sigma(\mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{xo}\mathbf{x}_t + \mathbf{b}_o) \tag{16}$$

$$\tag{17}$$

These gates control the flow of information inside the LSTM. They use a sigmoid activation (the σ) to produce a value between 0 and 1.

They behave like a gate or switch (0 is off and 1 is on).

Long-Short Term Memory

The hidden state is divided into two parts. $\mathbf{h}_t$ is state to predict output, and $\mathbf{C}_t$ is called cell state which models cross-timestep dependencies. First, a cell state proposal $\hat{\mathbf{C}}$ is computed.

$$\hat{\mathbf{C}} = \tanh(\mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{W}_{xc}\mathbf{x}_t + \mathbf{b}_c) \tag{18}$$

And then the final cell state $\mathbf{C}_t$ is updated using the forget $\mathbf{g}_f$ and input gates $\mathbf{g}_i$:

$$\mathbf{C}_t = \mathbf{g}_f \cdot \mathbf{C}_{t-1} + \mathbf{g}_i \cdot \hat{\mathbf{C}} \tag{19}$$

The forget gate determines how much of the previous cell state is used, and the input gate determines how the proposal is added to the final cell state.
The `tanh` activation produces values $\in [-1, 1]$, which intuitively can add/remove information in $\hat{\mathbf{C}}$.

Long-Short Term Memory

The output/hidden state is computed as:

$$\mathbf{h}_t = \mathbf{g}_o \cdot \sigma_y(\mathbf{C}_t) \tag{20}$$

Here the output gate controls which parts of the cell state are used as output, and σ_y is the activation function for the output (user configurable).

The LSTM is a complicated cell, but this additional complexity greatly helps model long-term dependencies and has less issues with vanishing/exploding gradients.

Long-Short Term Memory

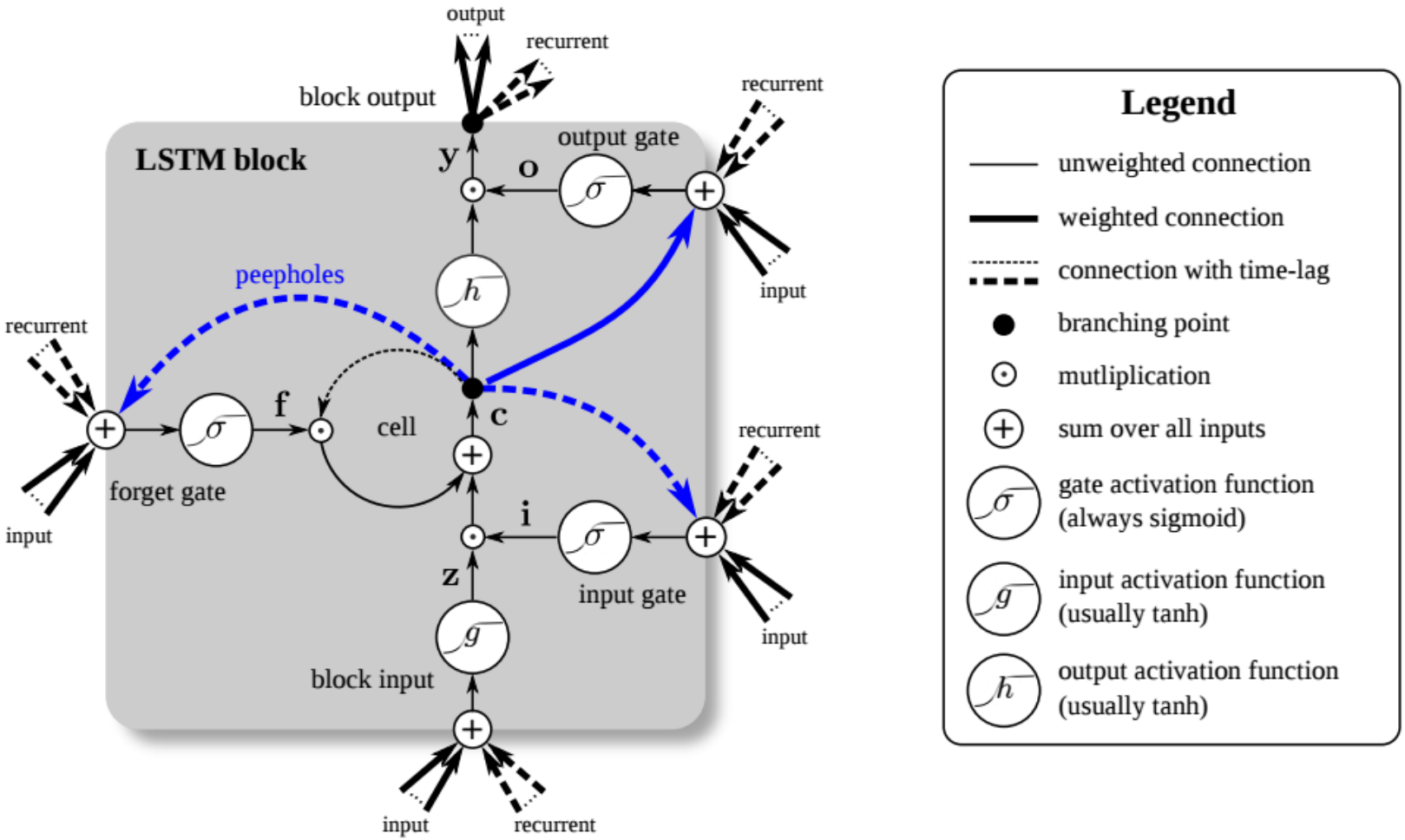


Figure: Graphical representation of the LSTM cell, with all the gates and their relationships.

Summary Neural Networks

Multi-Layer Perceptrons For general data, if flattened, specially tabular data or raw features.

Convolutional Networks For image data and matrix-data where there is local correlations between neighboring elements.

Recurrent Networks For sequence data like text, audio, time series, etc.

Of course note that these kinds of neural networks can be combined, you can build convolutional RNNs, etc.

What You Must Learn

This lecture contains a lot of content about three types of neural networks. What you should definitely learn for the future:

- (Three) Types of neural networks, their basic structure, concepts of forward pass, backward pass, autograd, weight sharing, types of layers, etc.
- Training with SGD, tuning learning rates, setting batch size.
- Issues with each type of neural network, what kind of data they can be applied to.

Book Chapter Readings

If you want to dive **deeper** in this topic, we recommend:

- Bishop Book: Chapter 5, in particulars Chapters 5.1-5.3, and 5.5.
- UDL Book: Chapter 3 for neural networks in general, Chapter 10 for convolutional networks.
- DL Book: Chapter 10 for recurrent neural networks.

UDL Book is freely available at
`https://udlbook.github.io/udlbook/`.
The DL book is freely available at
`https://www.deeplearningbook.org/`

What We Cover Later

Later we have another lecture on Deep Learning, which covers more Neural Network concepts including Deep Neural Networks. We will cover:

- Basic (convolutional) neural network architectures.
- More advanced layers (Dropout, Batch Normalization, etc).
- Sequence-to-Sequence models.
- Design of neural networks.

Questions to Think About

1. What are the differences between MLPs, CNNs, and RNNs?
2. Why do neural networks suffer from vanishing gradients?
3. You train a neural network in some regression dataset, and the training loss stays constant. What can be the issue and how do you solve it?
4. What are Long-Term Dependency problems and in which kind of NN do they happen?
5. What are the differences between a standard RNN and a LSTM?
6. What kind of neural network would you use to perform classification of videos?

Take Home Messages

- Neural networks are state of the art for many tasks in NLP, Computer Vision, etc.
- They are models built with interconnected layers that do feature learning or perform a task. Different layers are appropriate for different kinds of data and features.
- Neural Networks are trained using SGD, and since the loss is no longer convex, there are no guarantees of obtaining a global optima.
- Backpropagation is used to compute gradients analytically, exploiting the compositional nature of neural networks.

Questions?

Bibliography I

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

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