MATH 637: Mathematical Techniques in Data Science
Neural networks II

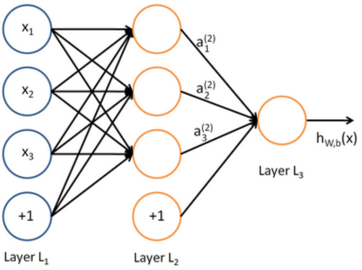
Dominique Guillot

Departments of Mathematical Sciences
University of Delaware

April 24, 2020

This lecture is based on the UFLDL tutorial (http://deeplearning.stanford.edu/tutorial/)

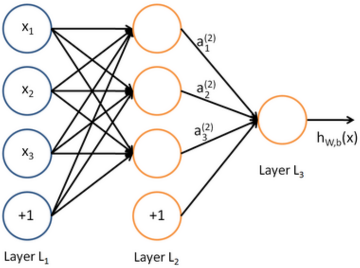
Recall



We have:

$$
\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b} &= a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}).
\end{aligned}
$$

Recall (cont.)



Vector form:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$
$$h_{W,b} = a^{(3)} = f(z^{(3)}).$$

Training neural networks

Suppose we have

- A neural network with s_l neurons in layer l ($l = 1, \ldots, n_l$).

Training neural networks

Suppose we have

- A neural network with s_l neurons in layer l ($l = 1, \dots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all l.

Training neural networks

Suppose we have

- A neural network with s_l neurons in layer l ($l = 1, \ldots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all l.

Let

$$J(W, b; x, y) := \frac{1}{2}\|h_{W,b}(x) - y\|_2^2 \qquad \text{(Squared error for one sample).}$$

Training neural networks

Suppose we have

- A neural network with s_l neurons in layer l ($l = 1, \ldots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all l.

Let

$$J(W, b; x, y) := \frac{1}{2}\|h_{W,b}(x) - y\|_2^2 \qquad \text{(Squared error for one sample)}.$$

Define

$$J(W, b) := \frac{1}{m}\sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2}\sum_{l=1}^{n_l - 1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2.$$

(average squared error with Ridge penalty).

Suppose we have

- A neural network with s_l neurons in layer l ($l = 1, \dots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all l.

Let

$$J(W, b; x, y) := \frac{1}{2}\|h_{W,b}(x) - y\|_2^2 \qquad \text{(Squared error for one sample).}$$

Define

$$J(W, b) := \frac{1}{m}\sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2.$$

(average squared error with Ridge penalty).

Note:

- The Ridge penalty prevents overfitting.

Suppose we have

- A neural network with s_l neurons in layer l ($l = 1, \dots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all l.

Let

$$J(W, b; x, y) := \frac{1}{2}\|h_{W,b}(x) - y\|_2^2 \qquad \text{(Squared error for one sample).}$$

Define

$$J(W, b) := \frac{1}{m}\sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2.$$

(average squared error with Ridge penalty).

Note:

- The Ridge penalty prevents overfitting.
- We do not penalize the bias terms $b_i^{(l)}$.

Training neural networks

Suppose we have

- A neural network with s_l neurons in layer l ($l = 1, \dots, n_l$).
- Observations $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)}) \in \mathbb{R}^{s_1} \times \mathbb{R}^{s_{n_l}}$.

We would like to choose $W^{(l)}$ and $b^{(l)}$ in some optimal way for all l.

Let

$$J(W, b; x, y) := \frac{1}{2}\|h_{W,b}(x) - y\|_2^2 \qquad \text{(Squared error for one sample)}.$$

Define

$$J(W, b) := \frac{1}{m}\sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) + \frac{\lambda}{2}\sum_{l=1}^{n_l - 1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2.$$

(average squared error with Ridge penalty).

Note:

- The Ridge penalty prevents overfitting.
- We do not penalize the bias terms $b_i^{(l)}$.
- The loss function $J(W, b)$ is not convex.

Some remarks

- The loss function $J(W, b)$ can be used both for regression and classification.

Some remarks

- The loss function $J(W, b)$ can be used both for regression and classification.
- Cross-entropy is frequently used for classification: measuring distance between probability distribution

$$
\begin{aligned}
y_i &= (1, 0, 0, 0) && \text{True label} \\
\hat{y}_i &= (0.63, 0.12, 0.07, 0.18) && \text{Predicted label.}
\end{aligned}
$$

Some remarks

- The loss function $J(W, b)$ can be used both for regression and classification.
- Cross-entropy is frequently used for classification: measuring distance between probability distribution

$$y_i = (1, 0, 0, 0) \qquad \text{True label}$$
$$\hat{y}_i = (0.63, 0.12, 0.07, 0.18) \qquad \text{Predicted label.}$$

- Scale the output according to the activation function in the last layer (e.g. $y \in [0, 1]$ if working with sigmoid).

Some remarks

- The loss function $J(W, b)$ can be used both for regression and classification.
- Cross-entropy is frequently used for classification: measuring distance between probability distribution

$$
\begin{aligned}
y_i &= (1, 0, 0, 0) && \text{True label} \\
\hat{y}_i &= (0.63, 0.12, 0.07, 0.18) && \text{Predicted label.}
\end{aligned}
$$

- Scale the output according to the activation function in the last layer (e.g. $y \in [0, 1]$ if working with sigmoid).
- The loss function is generally non-convex, we may only find a *local* minimum.

Some remarks

- The loss function $J(W, b)$ can be used both for regression and classification.
- Cross-entropy is frequently used for classification: measuring distance between probability distribution

$$y_i = (1, 0, 0, 0) \qquad \text{True label}$$
$$\hat{y}_i = (0.63, 0.12, 0.07, 0.18) \qquad \text{Predicted label.}$$

- Scale the output according to the activation function in the last layer (e.g. $y \in [0, 1]$ if working with sigmoid).
- The loss function is generally non-convex, we may only find a *local* minimum.
- We need an initial choice for $W_{ij}^{(l)}$ and $b_i^{(l)}$. If we initialize all the parameters to 0, then the parameters remain constant over the layers because of the symmetry of the problem.

Some remarks

- The loss function $J(W, b)$ can be used both for regression and classification.
- Cross-entropy is frequently used for classification: measuring distance between probability distribution

$$\begin{aligned} y_i &= (1, 0, 0, 0) && \text{True label} \\ \hat{y}_i &= (0.63, 0.12, 0.07, 0.18) && \text{Predicted label.} \end{aligned}$$

- Scale the output according to the activation function in the last layer (e.g. $y \in [0, 1]$ if working with sigmoid).
- The loss function is generally non-convex, we may only find a *local* minimum.
- We need an initial choice for $W_{ij}^{(l)}$ and $b_i^{(l)}$. If we initialize all the parameters to 0, then the parameters remain constant over the layers because of the symmetry of the problem.
- As a result, we usually initialize the parameters to a small constant at random (say, using $N(0, \epsilon^2)$ for $\epsilon = 0.01$).

Gradient descent and the backpropagation algorithm

- We update the parameters using a gradient descent as follows:

$$W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$

$$b_{i}^{(l)} \leftarrow b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} J(W, b).$$

- Here $\alpha > 0$ is a parameter (the *learning rate*).

Gradient descent and the backpropagation algorithm

- We update the parameters using a gradient descent as follows:

$$W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$

$$b_i^{(l)} \leftarrow b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b).$$

  - Here $\alpha > 0$ is a parameter (the *learning rate*).
- Observe that:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)}).$$

- We update the parameters using a gradient descent as follows:

$$W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$
$$b_{i}^{(l)} \leftarrow b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} J(W,b).$$

- Here $\alpha > 0$ is a parameter (the *learning rate*).
- Observe that:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) + \lambda W_{ij}^{(l)}$$
$$\frac{\partial}{\partial b_{i}^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_{i}^{(l)}} J(W,b;x^{(i)},y^{(i)}).$$

- Therefore, it suffices to compute the derivatives of $J(W,b;x^{(i)},y^{(i)})$.

Gradient descent and the backpropagation algorithm

- We update the parameters using a gradient descent as follows:

$$W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$

$$b_i^{(l)} \leftarrow b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b).$$

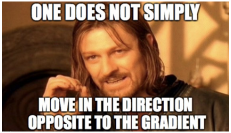
- Here $\alpha > 0$ is a parameter (the *learning rate*).
- Observe that:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)}).$$

- Therefore, it suffices to compute the derivatives of $J(W,b;x^{(i)},y^{(i)})$.
- The derivatives can be recursively computed using the chain rule (the backpropagation algorithm, or backprop). See Goodfellow et al. Section 6.5.

Stochastic gradient descent and minibatches

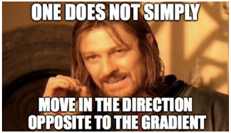


- The error to minimize has the form

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta)$$

for some loss function L and some vector of parameters θ ($n =$ sample size).

Stochastic gradient descent and minibatches



- The error to minimize has the form

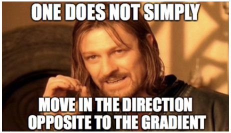
$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta)$$

for some loss function L and some vector of parameters θ ($n =$ sample size).

- Hence

$$\nabla_\theta J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

Stochastic gradient descent and minibatches



- The error to minimize has the form

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(x^{(i)}, y^{(i)}, \theta)$$

for some loss function L and some vector of parameters θ ($n =$ sample size).

- Hence

$$\nabla_\theta J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

- Thinking of $\nabla_\theta J(\theta)$ as an expected value:

$$\nabla_\theta J(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(x^{(t_i)}, y^{(t_i)}, \theta)$$

for a subset of samples $(x^{(t_1)}, y^{(t_1)}), \ldots, (x^{(t_m)}, y^{(t_m)})$ and $1 \leq m < n$.

Stochastic gradient descent and minibatches (cont.)

- Instead of computing the full gradient at every update, one can approximate it using random subset of samples of size $1 \leq m < n$.

Stochastic gradient descent and minibatches (cont.)

- Instead of computing the full gradient at every update, one can approximate it using random subset of samples of size $1 \leq m < n$.
- Typical approach: Divide the dataset into **minibatches** of a given size.
  1. Pick a minibatch.
  2. Approximate the gradient using that minibatch.
  3. Update the parameters of the model.
  4. Repeat Steps 1 to 3 until the whole dataset has been exhausted.

Stochastic gradient descent and minibatches (cont.)

- Instead of computing the full gradient at every update, one can approximate it using random subset of samples of size $1 \leq m < n$.
- Typical approach: Divide the dataset into **minibatches** of a given size.
  1. Pick a minibatch.
  2. Approximate the gradient using that minibatch.
  3. Update the parameters of the model.
  4. Repeat Steps 1 to 3 until the whole dataset has been exhausted.
- A complete pass through the dataset is called an **epoch**.

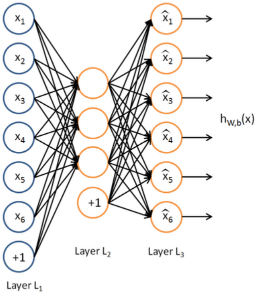
Stochastic gradient descent and minibatches (cont.)

- Instead of computing the full gradient at every update, one can approximate it using random subset of samples of size $1 \leq m < n$.
- Typical approach: Divide the dataset into **minibatches** of a given size.
  1. Pick a minibatch.
  2. Approximate the gradient using that minibatch.
  3. Update the parameters of the model.
  4. Repeat Steps 1 to 3 until the whole dataset has been exhausted.
- A complete pass through the dataset is called an **epoch**.
- The optimization process is often stopped after a given number of epochs.

Autoencoders

An **autoencoder** learns the identity function:

- Input: unlabeled data.
- Output = input.
- Idea: limit the number of hidden layers to discover structure in the data.
- Learn a *compressed* representation of the input.



Source: UFLDL tutorial.

Can also learn a *sparse* network by including supplementary constraints on the weights.

Example (UFLDL)

- Train an autoencoder on 10×10 images with one hidden layer.

Example (UFLDL)

- Train an autoencoder on 10×10 images with one hidden layer.
- Each hidden unit i computes:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_j^{(1)}\right).$$

Example (UFLDL)

- Train an autoencoder on 10×10 images with one hidden layer.
- Each hidden unit i computes:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_j^{(1)}\right).$$

- Think of $a_i^{(2)}$ as some non-linear feature of the input x.

Example (UFLDL)

- Train an autoencoder on 10×10 images with one hidden layer.
- Each hidden unit i computes:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_j^{(1)}\right).$$

- Think of $a_i^{(2)}$ as some non-linear feature of the input x.

Problem: Find x that maximally activates $a_i^{(2)}$ over $\|x\|_2 \leq 1$.

Example (UFLDL)

- Train an autoencoder on 10×10 images with one hidden layer.
- Each hidden unit i computes:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_j^{(1)}\right).$$

- Think of $a_i^{(2)}$ as some non-linear feature of the input x.

Problem: Find x that maximally activates $a_i^{(2)}$ over $\|x\|_2 \leq 1$.

Claim:

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$

Example (UFLDL)

- Train an autoencoder on 10×10 images with one hidden layer.
- Each hidden unit i computes:

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_j^{(1)}\right).$$

- Think of $a_i^{(2)}$ as some non-linear feature of the input x.

Problem: Find x that maximally activates $a_i^{(2)}$ over $\|x\|_2 \leq 1$.

Claim:

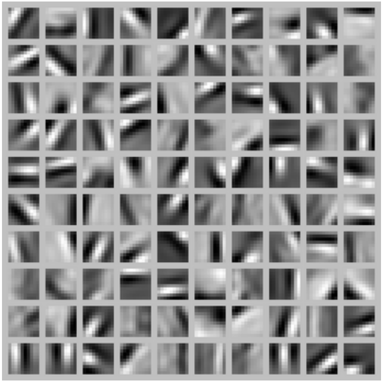
$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}.$$

(Hint: Use Cauchy–Schwarz).

We can now display the image maximizing $a_i^{(2)}$ for each i.

Example (cont.)

100 hidden units on 10x10 pixel inputs:



The different hidden units have learned to detect edges at different positions and orientations in the image.

Sparse neural networks

- So far we discussed *dense* neural networks.

Sparse neural networks

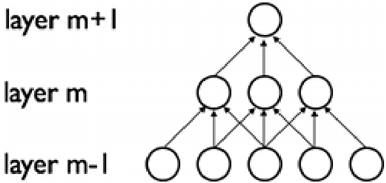
- So far we discussed *dense* neural networks.
- Dense networks have a lot of parameters to learn. Can be inefficient or impossible to train.

Sparse neural networks

- So far we discussed *dense* neural networks.
- Dense networks have a lot of parameters to learn. Can be inefficient or impossible to train.
- *Sparse* models have been proposed in the literature.

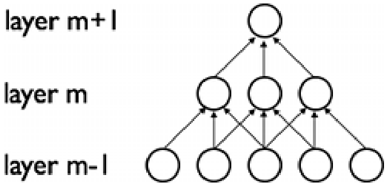
Sparse neural networks

- So far we discussed *dense* neural networks.
- Dense networks have a lot of parameters to learn. Can be inefficient or impossible to train.
- *Sparse* models have been proposed in the literature.
- Some of these models find inspiration from how the early visual system is wired up in biology.



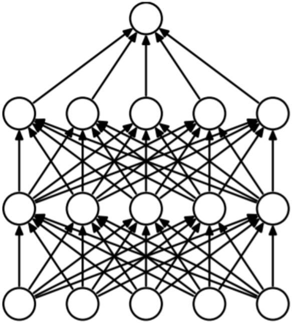
Sparse neural networks

- So far we discussed *dense* neural networks.
- Dense networks have a lot of parameters to learn. Can be inefficient or impossible to train.
- *Sparse* models have been proposed in the literature.
- Some of these models find inspiration from how the early visual system is wired up in biology.

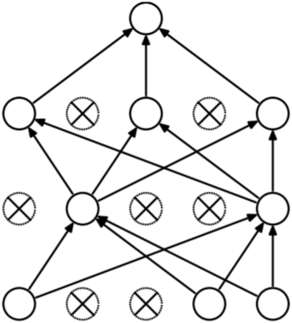


- **Dropouts:** During training, randomly ignore or ("drop out") some neurons.
- Can specify a dropout *rate* (i.e., a fixed probability $0 \leq p \leq 1$ of ignoring a given node).
- Used to learn sparse models and prevent overfitting.

Dropouts



(a) Standard Neural Net



(b) After applying dropout.

Using convolutions

- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.

Using convolutions

- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.

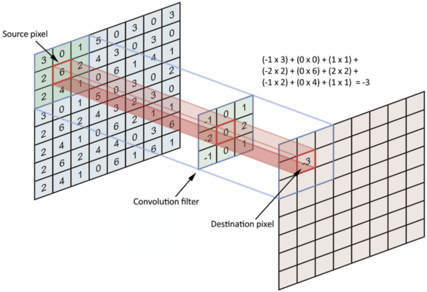
Using convolutions

- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
- That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.

Using convolutions

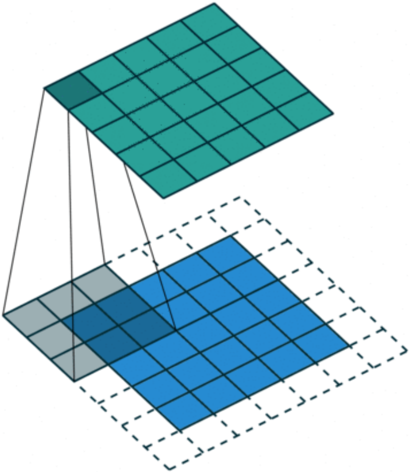
- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
- That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.
- Can "convolve" the learned features with the larger image.

- Idea: Certain signals are *stationary*, i.e., their statistical properties do not change in space or time.
- For example, images often have similar statistical properties in different regions in space.
- That suggests that the features that we learn at one part of an image can also be applied to other parts of the image.
- Can "convolve" the learned features with the larger image.

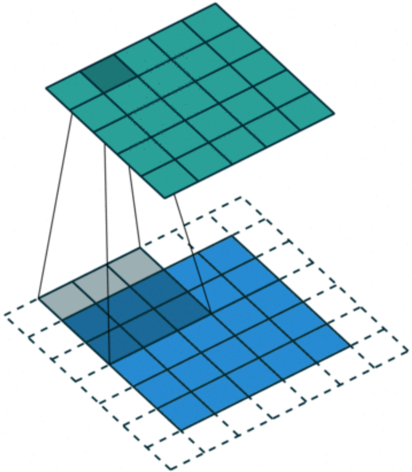


$$C(x,y) = \sum_m \sum_n I(x+m, y+m) K(m,n).$$

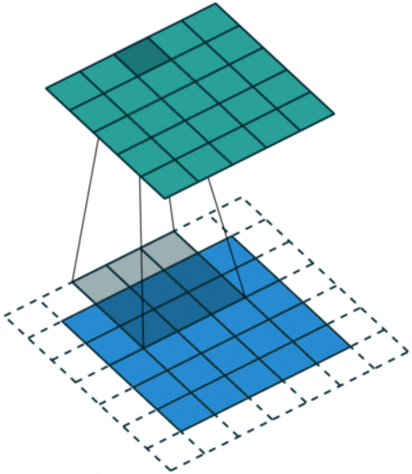
Example



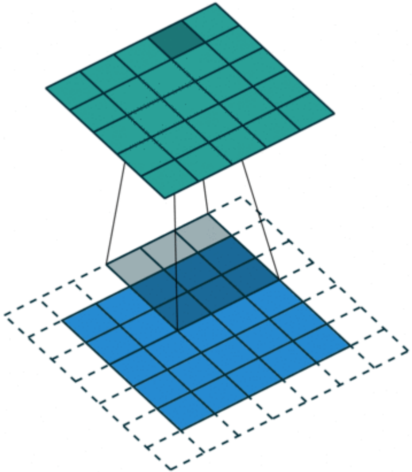
Example



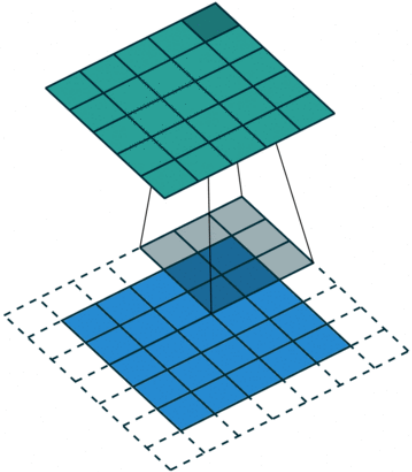
Example

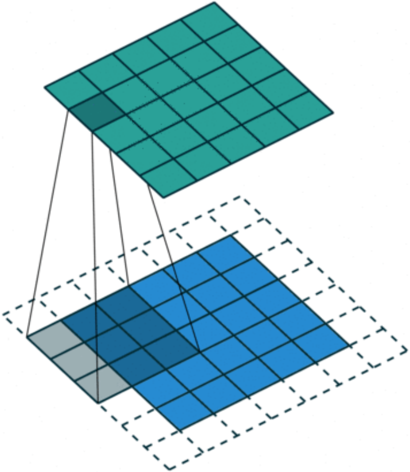


Example

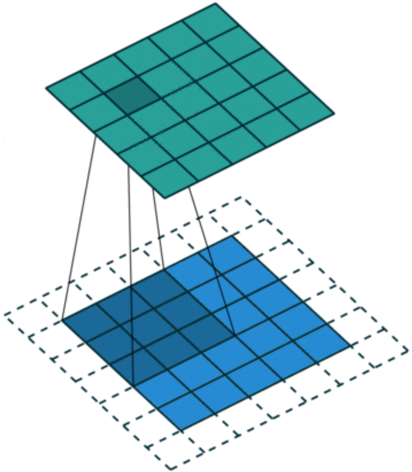


Example

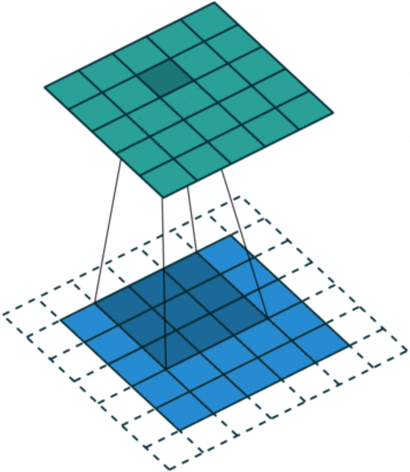




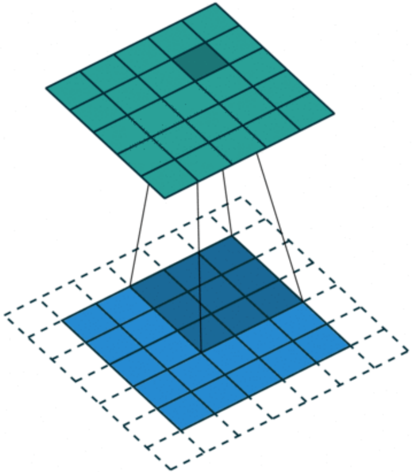
Example



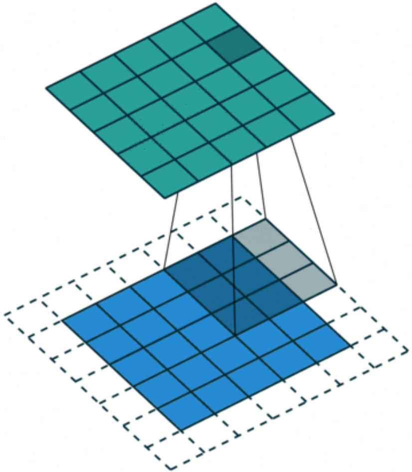
Example



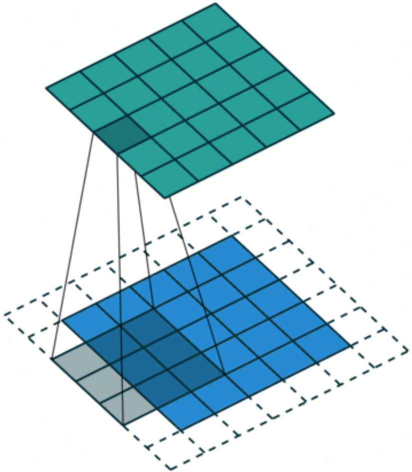
Example



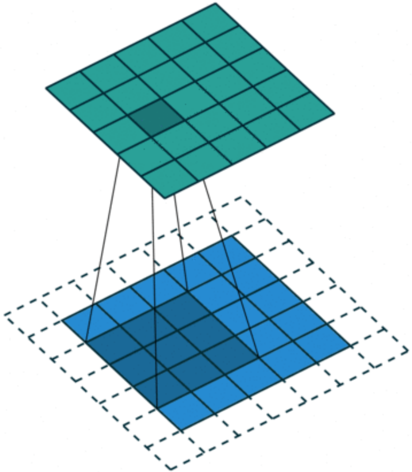
Example



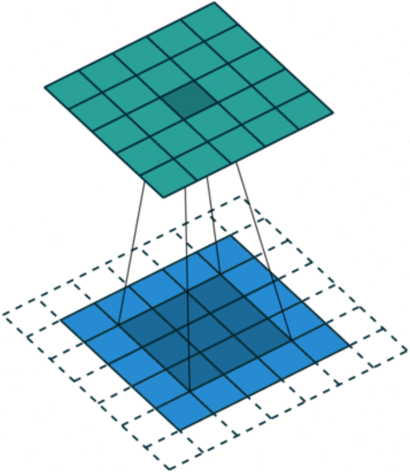
Example



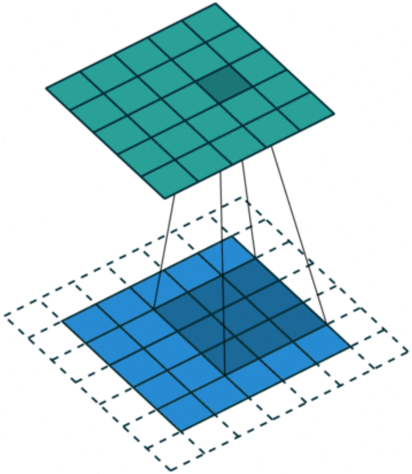
Example



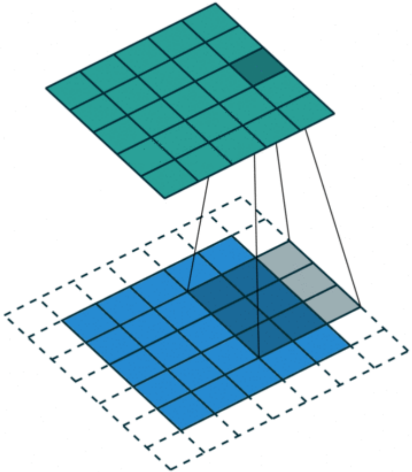
Example



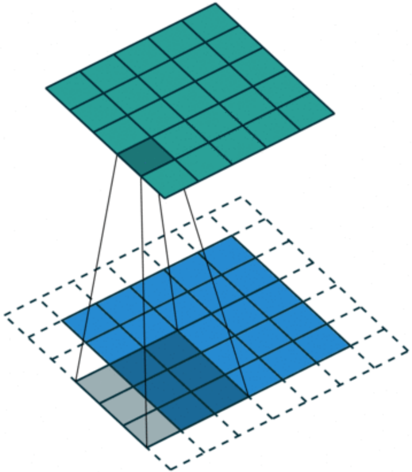
Example



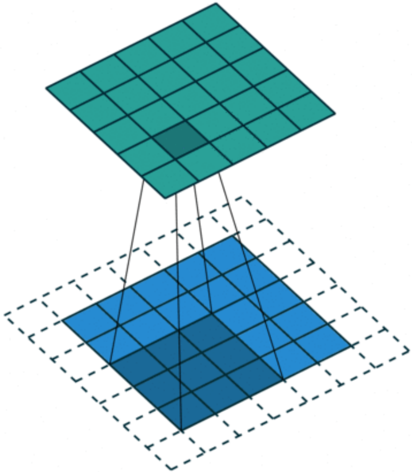
Example



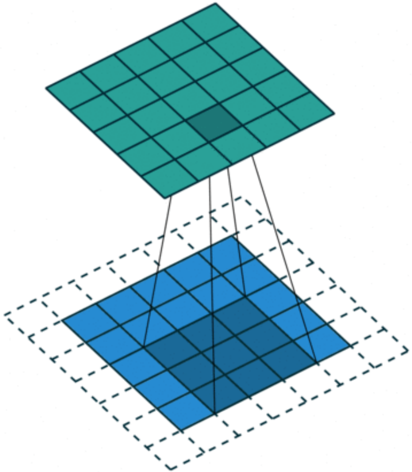
Example



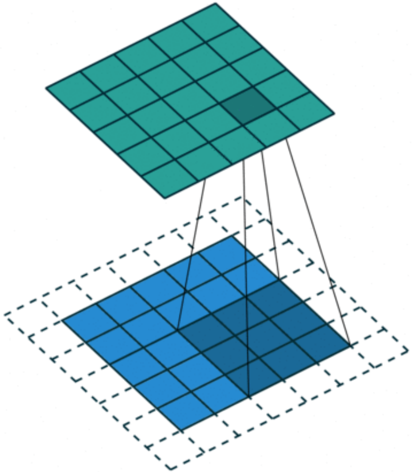
Example



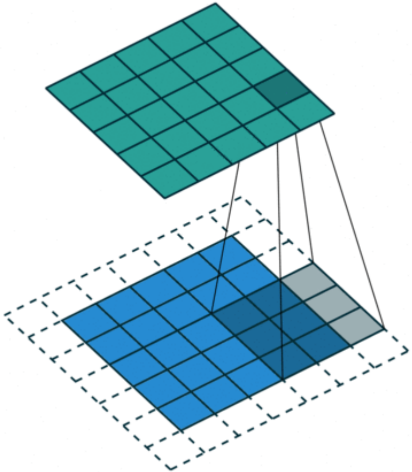
Example



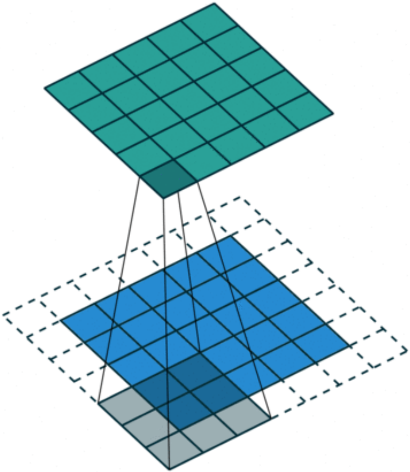
Example



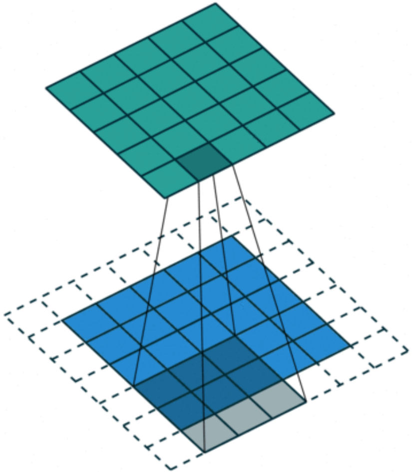
Example



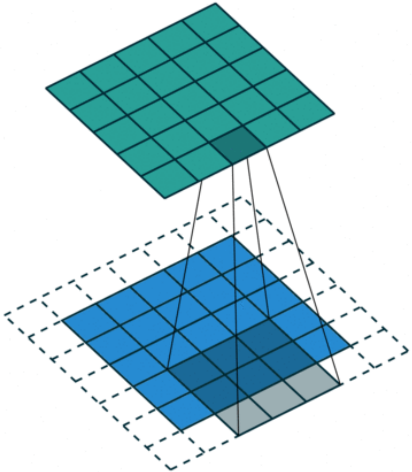
Example



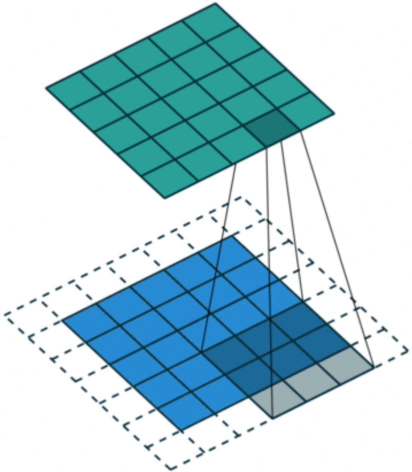
Example



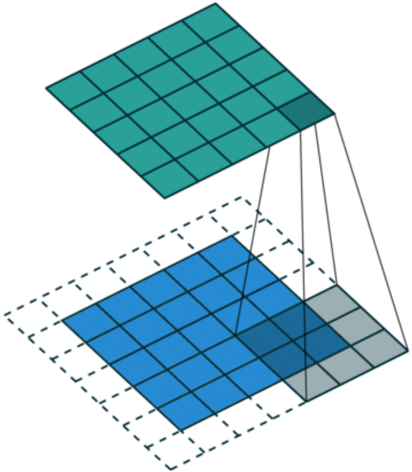
Example



Example



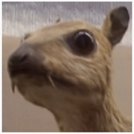
Example



Filters

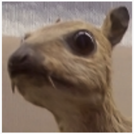
Kernel

Edge detection

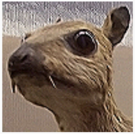


$$* \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} =$$

Sharpen

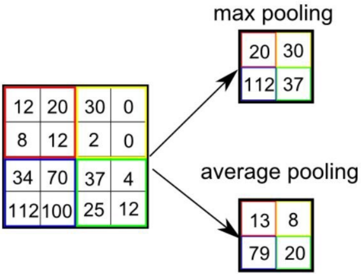


$$* \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} =$$

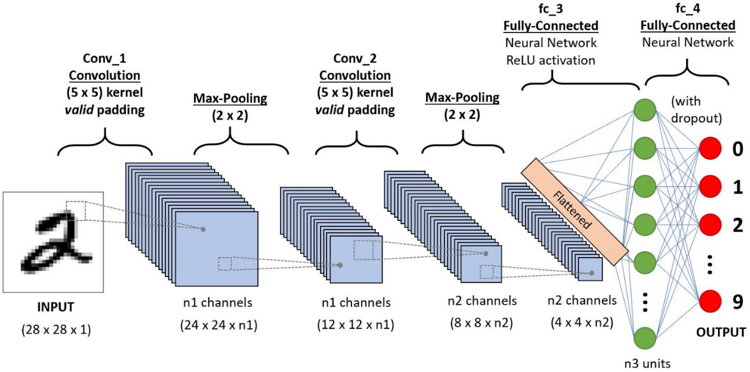


Pooling features

- Once can also *pool* the features obtained via convolution (max, mean, etc.).
- Can lead to more robust features. Can lead to invariant features.
- For example, if the pooling regions are contiguous, then the pooling units will be "translation invariant", i.e., they won't change much if objects in the image are undergo a (small) translation.



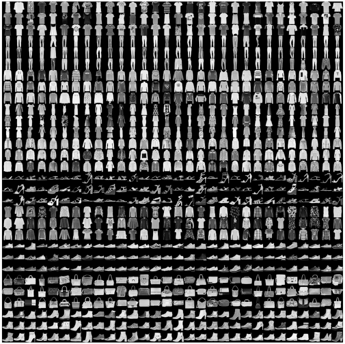
Example: handwritten digits



Homework.

Please go through (and run on your own) the tutorial available at:

`https://www.tensorflow.org/tutorials/keras/classification`



- If using Anaconda: `conda install tensorflow`.
- Can also use Google Colab:

`https://colab.research.google.com/`