Spatial Transformer Networks — Backpropagation

A Self-Contained Introduction

Thomas Kurbiel · Following

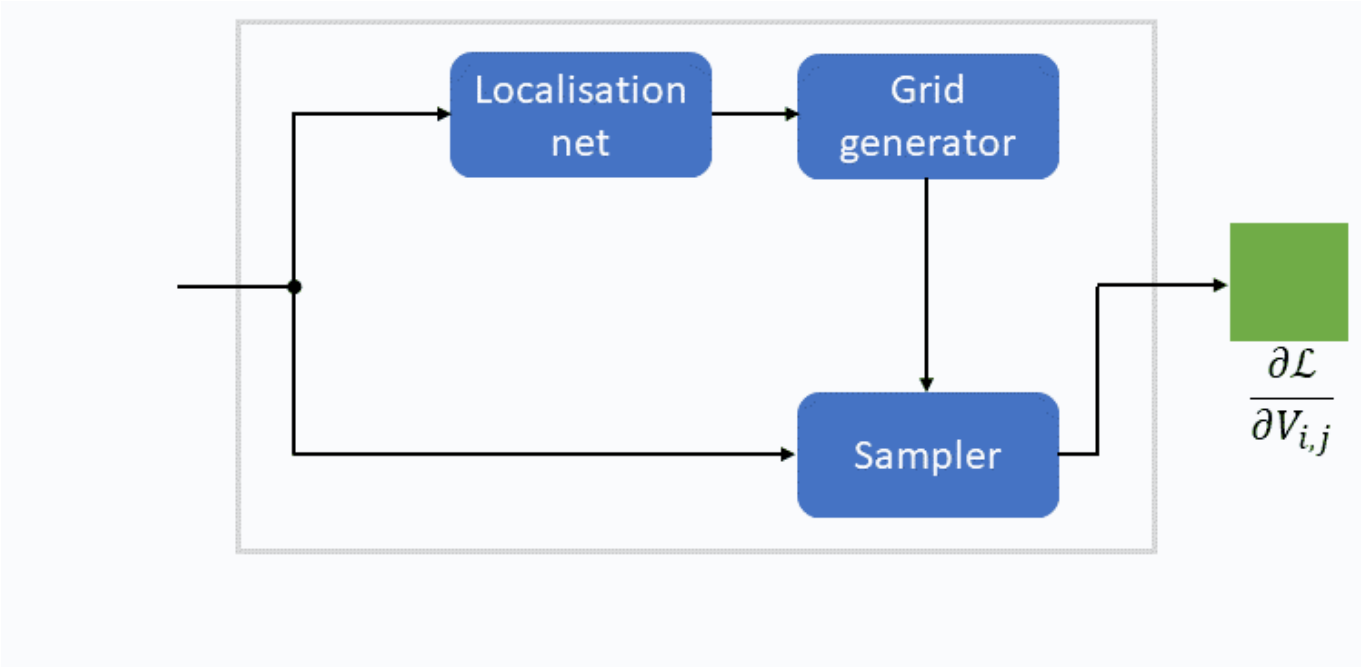
Published in Towards Data Science · 8 min read · Oct 12, 2021

Spatial Transformer modules, introduced by Max Jaderberg et al., are a popular way to increase spatial invariance of a model against spatial transformations such as translation, scaling, rotation, cropping, as well as non-rigid deformations. They achieve spatial invariance by adaptively transforming their input to a canonical, expected pose, thus leading to a better classification performance.

In this four-part tutorial, we cover all prerequisites needed for gaining a deep understanding of spatial transformers. In the first two posts, we have introduced the concepts of forward and reverse mapping and delved into the details of bilinear interpolation. In the last post, we have introduced all building blocks a spatial transformer module is made of. Finally, in this post, we will derive all necessary backpropagation equations from scratch.

Gradient Flow

Before we start deriving formulas, let us quickly take a look at how the gradient is flowing back through a spatial transformer module:



Gradient flow, $\mathcal{L}$ denotes the loss function (Image by author)

The animation above clearly illustrates why spatial transformers networks can be trained end-to-end using standard backpropagation. We start with the gradient at the output of the module, which already has been computed in a higher layer. The first thing we have to do, is to derive explicit formulas to propagate (or flow) this gradient back through the **sampler** to both the input feature map and the **sampling grid**. Than we will have to derive the formulas governing the backpropagation through the **grid generator**. Remember that **sampler** and **grid generator** are both parameter less operations, i.e. don't have any trainable parameters. Lastly, we have to backpropagate the gradient through the **localisation network**, which is a standard neural network, hence no new formulas have to be derived here. It is the **localisation network** where a parameter update is taking place.

If you have never come across backpropagation and have problems to grasp the concept of gradient flow, please take a look at my introductory post.

Gradient w.r.t sampling grid coordinates

As in all previous posts we will assume the **sampler** is using bilinear interpolation to transform the input feature map. Let us quickly recall the corresponding formula, which we derived in the second post. For each entry of the **sampling grid**:

$$(y^s_{i,j}, x^s_{i,j})$$

the **sampler** first finds the four neighboring values by taking the floor and ceil operations:

$$\text{upper left: } U_{\underline{y_{i,j}}, \underline{x_{i,j}}} \qquad \text{upper right: } U_{\underline{y_{i,j}}, \overline{x_{i,j}}}$$
$$\text{lower left: } U_{\overline{y_{i,j}}, \underline{x_{i,j}}} \qquad \text{lower right: } U_{\overline{y_{i,j}}, \overline{x_{i,j}}}$$

We dropped the superscript "*s*" for the sake of clarity. Next, the **sampler** calculates the *horizontal distance* from the sample point to its right cell border and the *vertical distance* to the top cell border:

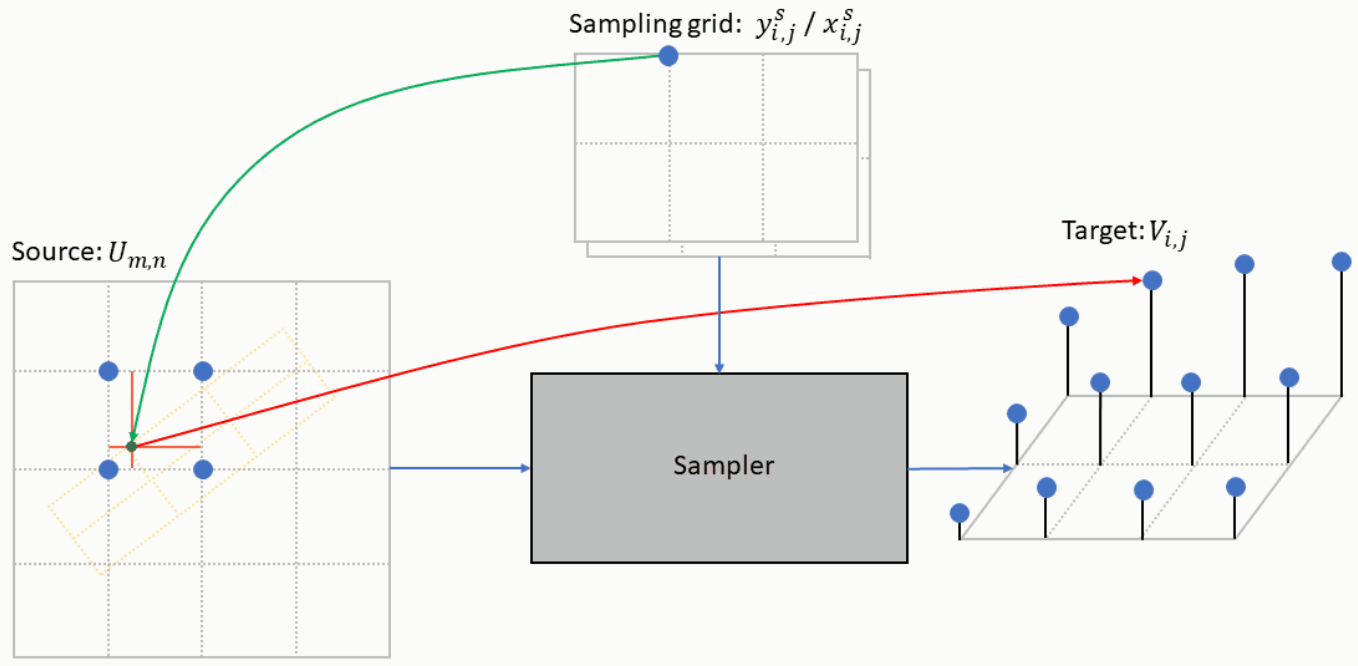
$$d_{x_{i,j}} = \overline{x_{i,j}} - x_{i,j} \qquad \text{and} \qquad d_{y_{i,j}} = \overline{y_{i,j}} - y_{i,j}$$

Finally, it takes a weighted average to produce the output:

$$V_{i,j} = U_{\underline{y_{i,j}},\underline{x_{i,j}}} \cdot d_{y_{i,j}} \cdot d_{x_{i,j}} + U_{\underline{y_{i,j}},\overline{x_{i,j}}} \cdot d_{y_{i,j}} \cdot (1 - d_{x_{i,j}}) +$$

$$U_{\overline{y_{i,j}},\underline{x_{i,j}}} \cdot (1 - d_{y_{i,j}}) \cdot d_{x_{i,j}} + U_{\overline{y_{i,j}},\overline{x_{i,j}}} \cdot (1 - d_{y_{i,j}}) \cdot (1 - d_{x_{i,j}})$$

To get an intuition for the wanted derivatives, let us wiggle a single (!) entry of the **sampling grid**:



Impact of a single entry on the output map (Image by author)

We see, that the wiggling affects only a single pixel in the output feature map. This is to be expected, since the **sampler** operates independently on each entry of the **sampling grid** (the reason why the sampler lends itself perfectly to parallelization). To backpropagate the loss error from the output feature map to the **sampling grid**, all we have to do is to apply the chain rule:

$$\frac{\partial \mathcal{L}}{\partial x_{i,j}} = \frac{\partial \mathcal{L}}{\partial V_{i,j}} \cdot \frac{\partial V_{i,j}}{\partial x_{i,j}} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial y_{i,j}} = \frac{\partial \mathcal{L}}{\partial V_{i,j}} \cdot \frac{\partial V_{i,j}}{\partial y_{i,j}}$$

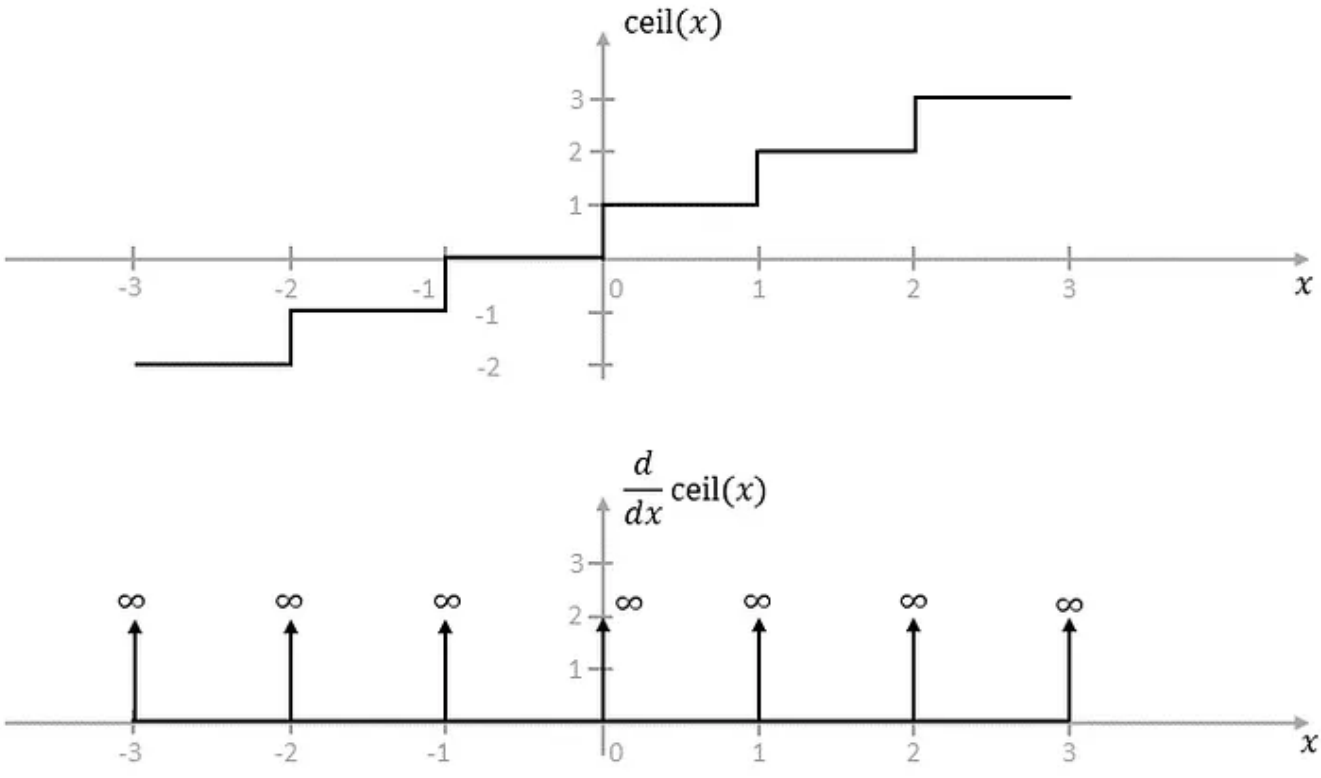
where $\mathcal{L}$ is the loss function. Next, we must take the derivative of V w.r.t to x:

$$\frac{\partial V_{i,j}}{\partial x_{i,j}} = U_{\underline{y_{i,j},x_{i,j}}} \cdot d_{y_{i,j}} \cdot \frac{\partial d_{x_{i,j}}}{\partial x_{i,j}} + U_{\underline{y_{i,j}},\overline{x_{i,j}}} \cdot d_{y_{i,j}} \cdot \frac{\partial (1 - d_{x_{i,j}})}{\partial x_{i,j}} + U_{\overline{y_{i,j}},\underline{x_{i,j}}} \cdot (1 - d_{y_{i,j}}) \cdot \frac{\partial d_{x_{i,j}}}{\partial x_{i,j}} + U_{\overline{y_{i,j}},\overline{x_{i,j}}} \cdot (1 - d_{y_{i,j}}) \cdot \frac{\partial (1 - d_{x_{i,j}})}{\partial x_{i,j}}$$

which requires us to take the derivative of the horizontal distance:

$$\frac{\partial d_x}{\partial x} = \frac{\partial (\overline{x} - x)}{\partial x} = \frac{\partial \overline{x}}{\partial x} - \frac{\partial x}{\partial x}$$

To proceed further we have to take a look at the derivative of the ceiling operation:



Ceiling operation and its derivative (Image by author)

As we can see, the ceiling operation is piecewise constant and the derivative of a constant is zero. The ceiling operation is discontinuous at integer values of x and is non-differentiable there.

Technically speaking we cannot apply gradient descent on a non-differentiable function. Our remedy is the so-called sub-derivative, which is an extension of the derivative, see *references*. Practically it boils down to setting the derivative to zero at integer values of x:

$$\frac{\partial d_x}{\partial x} = \frac{\partial \overline{x}}{\partial x} - \frac{\partial x}{\partial x} \approx 0 - 1 = -1$$

and analogously:

$$\frac{\partial (1 - d_x)}{\partial x} = \frac{\partial 1}{\partial x} - \frac{\partial d_x}{\partial x} = 1$$

Formally, we are now calculating the sub-gradients instead of gradients. Our final formula is:

$$\frac{\partial V_{i,j}}{\partial x_{i,j}} = -U_{\underline{y_{i,j}},\underline{x_{i,j}}} \cdot d_{y_{i,j}} + U_{\underline{y_{i,j}},\overline{x_{i,j}}} \cdot d_{y_{i,j}} - U_{\overline{y_{i,j}},\underline{x_{i,j}}} \cdot (1 - d_{y_{i,j}}) + U_{\overline{y_{i,j}},\overline{x_{i,j}}} \cdot (1 - d_{y_{i,j}})$$

and after rearranging:

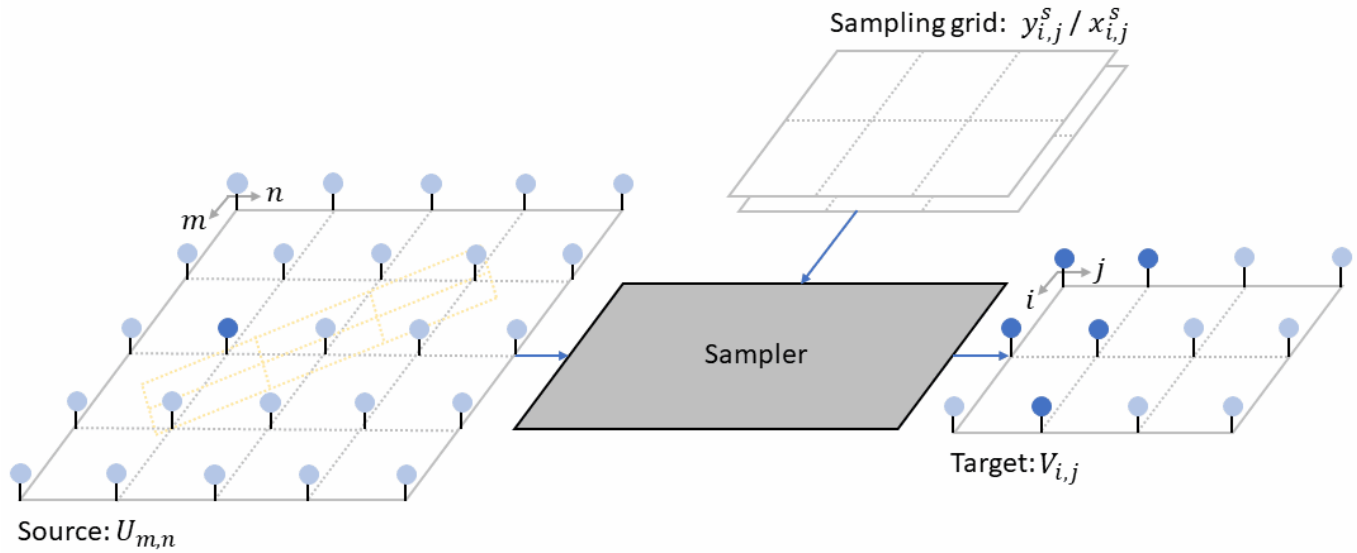
$$\frac{\partial V_{i,j}}{\partial x_{i,j}} = (1 - d_{y_{i,j}}) \cdot (U_{\overline{y_{i,j}},\overline{x_{i,j}}} - U_{\overline{y_{i,j}},\underline{x_{i,j}}}) + d_{y_{i,j}} \cdot (U_{\underline{y_{i,j}},\overline{x_{i,j}}} - U_{\underline{y_{i,j}},\underline{x_{i,j}}})$$

For the y component we get accordingly:

$$\frac{\partial V_{i,j}}{\partial y_{i,j}} = (1 - d_{x_{i,j}}) \cdot (U_{\overline{y_{i,j}},\overline{x_{i,j}}} - U_{\underline{y_{i,j}},\overline{x_{i,j}}}) + d_{x_{i,j}} \cdot (U_{\overline{y_{i,j}},\underline{x_{i,j}}} - U_{\underline{y_{i,j}},\underline{x_{i,j}}})$$

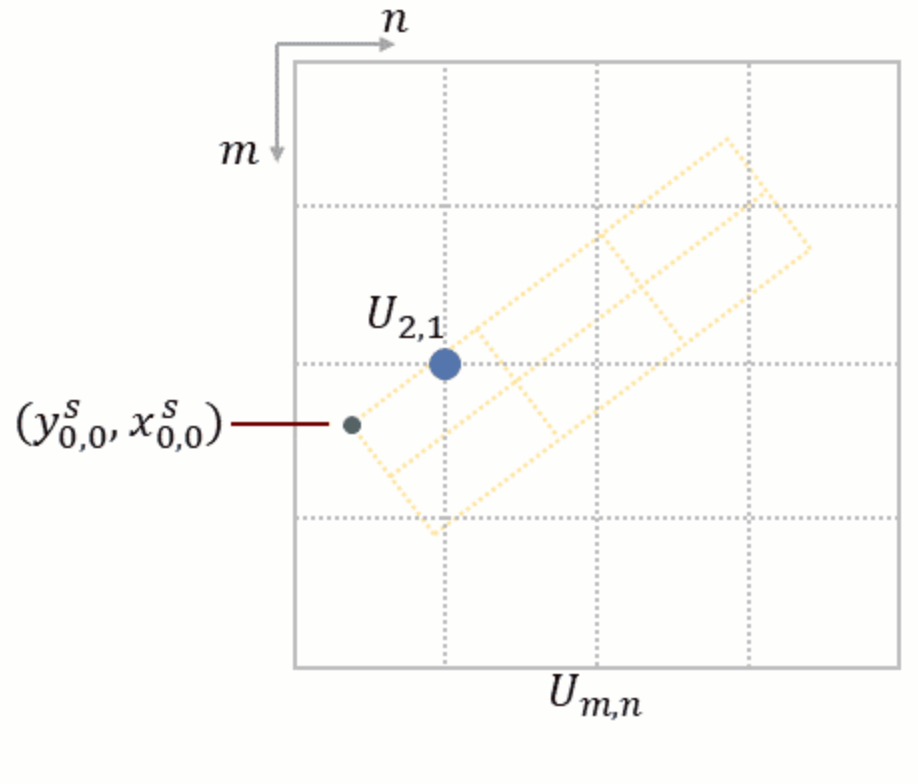
Gradient w.r.t input feature map

Before we dive into mathematical formulas, let us again first develop an intuition. This time we must wiggle the value of a pixel in the input feature map, say at coordinates (2, 1):



Impact of a single input pixel on the output map (Image by author)

We see, that wiggling a single pixel in the input feature map, causes several pixels in the output feature map to change. To understand the reason, let us take a closer look at the sample points of the affected output pixels:



Affected sample points share a common input pixel (Image by author)

We notice that all mentioned sample points have something in common: the input pixel at coordinates (2, 1) always belongs to one of their four neighboring points used in bilinear interpolation. Please also notice how input pixel (2, 1) is sometimes the upper right neighbor, sometimes lower left neighbor and so on.

The chain rule used to backpropagate the error now becomes:

$$\frac{\partial \mathcal{L}}{\partial U_{m,n}} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} \frac{\partial \mathcal{L}}{\partial V_{i,j}} \cdot \frac{\partial V_{i,j}}{\partial U_{m,n}}$$

where the two sums consider the fact, that each pixel in the input feature map might (potentially) affect multiple pixels in the output feature map. In the next step, we must evaluate the expression $\partial V / \partial U$, which strongly

depends on the relative position of U with respect to V's sample point (upper left neighbor, upper right neighbor etc.). To this end we rewrite the bilinear interpolation formula in the following way:

$$V_{i,j} = U_{\text{UL}} \cdot d_{y_{i,j}} \cdot d_{x_{i,j}} + U_{\text{UR}} \cdot d_{y_{i,j}} \cdot (1 - d_{x_{i,j}}) + U_{\text{LL}} \cdot (1 - d_{y_{i,j}}) \cdot d_{x_{i,j}} + U_{\text{LR}} \cdot (1 - d_{y_{i,j}}) \cdot (1 - d_{x_{i,j}})$$

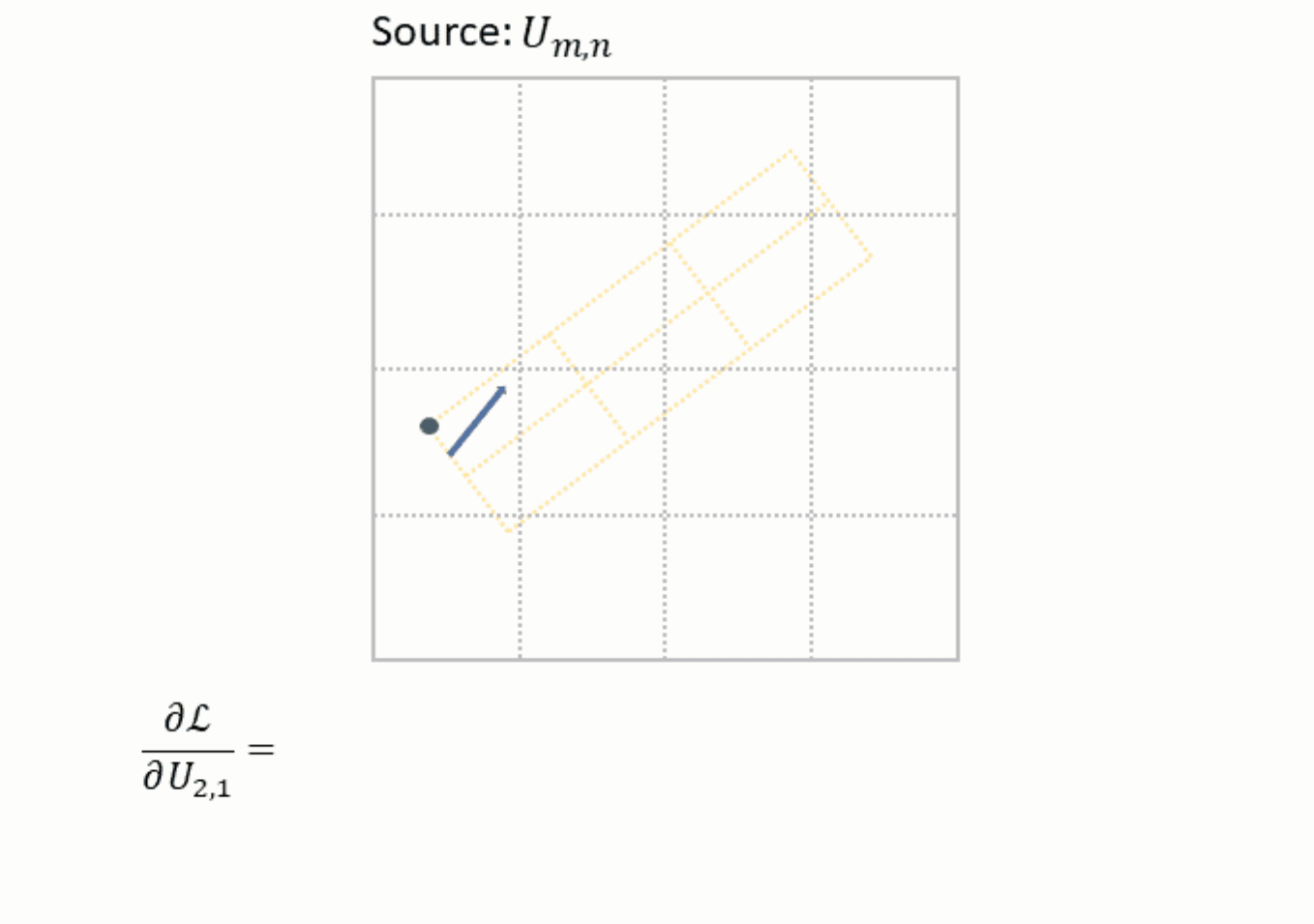
The corresponding derivatives are:

$$\begin{aligned} &\text{upper left:} && \partial V_{i,j}/\partial U_{\text{UL}} = d_{y_{i,j}} \cdot d_{x_{i,j}} \\ &\text{upper right:} && \partial V_{i,j}/\partial U_{\text{UR}} = d_{y_{i,j}} \cdot (1 - d_{x_{i,j}}) \\ &\text{lower left:} && \partial V_{i,j}/\partial U_{\text{LL}} = (1 - d_{y_{i,j}}) \cdot d_{x_{i,j}} \\ &\text{lower right:} && \partial V_{i,j}/\partial U_{\text{LR}} = (1 - d_{y_{i,j}}) \cdot (1 - d_{x_{i,j}}) \end{aligned}$$

We now have all the necessary formulas to compute the gradient. To get a better intuition for the whole procedure, let us apply it to the example in the animation above. Here, input pixel (2, 1) affects the following five output pixels (0, 0), (0, 1), (1, 0), (1, 1) and (2, 1):

$$\begin{aligned} \text{upper right} \quad & \frac{\partial \mathcal{L}}{\partial U_{2,1}} = \frac{\partial \mathcal{L}}{\partial V_{0,0}} \cdot \frac{\partial V_{0,0}}{\partial U_{2,1}} + \\ \text{lower left} \quad & \frac{\partial \mathcal{L}}{\partial V_{0,1}} \cdot \frac{\partial V_{0,1}}{\partial U_{2,1}} + \\ \text{upper right} \quad & \frac{\partial \mathcal{L}}{\partial V_{1,0}} \cdot \frac{\partial V_{1,0}}{\partial U_{2,1}} + \\ \text{upper left} \quad & \frac{\partial \mathcal{L}}{\partial V_{1,1}} \cdot \frac{\partial V_{1,1}}{\partial U_{2,1}} + \\ \text{upper left} \quad & \frac{\partial \mathcal{L}}{\partial V_{2,1}} \cdot \frac{\partial V_{2,1}}{\partial U_{2,1}} \end{aligned} \quad \Rightarrow \quad \begin{aligned} & \frac{\partial \mathcal{L}}{\partial U_{2,1}} = \frac{\partial \mathcal{L}}{\partial V_{0,0}} \cdot d_{y_{0,0}} \cdot (1 - d_{x_{0,0}}) + \\ & \frac{\partial \mathcal{L}}{\partial V_{0,1}} \cdot (1 - d_{y_{0,1}}) \cdot d_{x_{0,1}} + \\ & \frac{\partial \mathcal{L}}{\partial V_{1,0}} \cdot d_{y_{1,0}} \cdot (1 - d_{x_{1,0}}) + \\ & \frac{\partial \mathcal{L}}{\partial V_{1,1}} \cdot d_{y_{1,1}} \cdot d_{x_{1,1}} + \\ & \frac{\partial \mathcal{L}}{\partial V_{2,1}} \cdot d_{y_{2,1}} \cdot d_{x_{2,1}} \end{aligned}$$

The main challenge of the procedure seems to lie in finding all the affected output pixels. Luckily, in the actual implementation we can omit the explicit search altogether by exploiting linearity:



Efficient implementation (Image by author)

To this end we first initialize an empty array for the gradient $\partial \mathcal{L} / \partial U$ and then iterate over the entries of the **sampling grid**. For each entry we use the second last formula to compute all four derivatives $\partial V / \partial U$, which we subsequently multiply by the corresponding entry of the gradient $\partial \mathcal{L} / \partial V$. The last remaining step, is to add the four computed values to the gradient array. Please note, that each value is added at a different position, defined by the positions of the four neighboring points. At the end of the

whole procedure, each entry of the gradient array will contain a complete sum of all affected output pixels.

Backpropagating through the grid generator

We have seen how the loss function depends on all the coordinates of the **sampling grid**:

$$\mathcal{L}(y^s_{0,0}, x^s_{0,0}, \quad y^s_{0,1}, x^s_{0,1}, \quad \dots \quad , y^s_{i,j}, x^s_{i,j}, \quad \dots \quad)$$

Furthermore, each sample coordinate is a function of the parameters provided by the **localisation network**:

$$y^s_{i,j} = f_y(\theta_1, \dots, \theta_K), \qquad x^s_{i,j} = f_x(\theta_1, \dots, \theta_K)$$

Applying the chain rule for multivariate functions hence gives us:

$$\frac{\partial \mathcal{L}}{\partial \theta_k} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} \left(\frac{\partial \mathcal{L}}{\partial y^s_{i,j}} \cdot \frac{\partial y^s_{i,j}}{\partial \theta_k} + \frac{\partial \mathcal{L}}{\partial x^s_{i,j}} \cdot \frac{\partial x^s_{i,j}}{\partial \theta_k} \right)$$

In the following, we will assume the **grid generator** is using an affine transformation:

$$x^s_{i,j} = \theta_1 \cdot x^t_{i,j} + \theta_2 \cdot y^t_{i,j} + \theta_5$$

$$y^s_{i,j} = \theta_3 \cdot x^t_{i,j} + \theta_4 \cdot y^t_{i,j} + \theta_6$$

Since the target coordinates lie on a regular sampling grid, we have:

$$y^t_{i,j} = i \quad \text{and} \quad x^t_{i,j} = j$$

such that the above equation reduces to:

$$x^s_{i,j} = \theta_1 \cdot j + \theta_2 \cdot i + \theta_5$$

$$y^s_{i,j} = \theta_3 \cdot j + \theta_4 \cdot i + \theta_6$$

The corresponding derivatives are:

$$\frac{\partial x^s_{i,j}}{\partial \theta_1} = j \quad \text{and} \quad \frac{\partial x^s_{i,j}}{\partial \theta_2} = i \quad \text{and} \quad \frac{\partial x^s_{i,j}}{\partial \theta_5} = 1$$

$$\frac{\partial y^s_{i,j}}{\partial \theta_3} = j \quad \text{and} \quad \frac{\partial y^s_{i,j}}{\partial \theta_4} = i \quad \text{and} \quad \frac{\partial y^s_{i,j}}{\partial \theta_6} = 1$$

and zero for the remaining cases. To obtain the final formulas, we have plug those derivatives back into the chain rule above:

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} j \cdot \frac{\partial \mathcal{L}}{\partial x^s_{i,j}} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \theta_2} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} i \cdot \frac{\partial \mathcal{L}}{\partial x^s_{i,j}} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \theta_5} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} \frac{\partial \mathcal{L}}{\partial x^s_{i,j}}$$

and analogously:

$$\frac{\partial \mathcal{L}}{\partial \theta_3} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} j \cdot \frac{\partial \mathcal{L}}{\partial y^s_{i,j}} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \theta_4} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} i \cdot \frac{\partial \mathcal{L}}{\partial y^s_{i,j}} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial \theta_6} = \sum_{i=0}^{H'-1} \sum_{j=0}^{W'-1} \frac{\partial \mathcal{L}}{\partial y^s_{i,j}}$$

As mentioned in the first section, the **grid generator** is usually implemented as a standard neural network, such as a fully connected network or a convolutional network. For this reason, we don't need to derive any new backpropagation formulas.

We have come to the end of the fourth and last post. In this post we have covered the rarely addressed topic of backpropagation in a spatial transformer module. Though complicated, it will significantly help you to debug issues, potentially faced when working with spatial transformer modules. If you want to hone your understanding further, please take a look at the actual Tensorflow C++ implementation of backpropagation through the sampler.

References

Original Paper
Beyond the derivative — subderivatives
ReLU : Not a Differentiable Function
Subderivative

Deep Learning | Machine Learning | Data Science | Computer Vision | Artificial Intelligence

More from the list: "NLP"

Curated by Himanshu Birla

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

· 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

· 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

· 15 min rea

View list

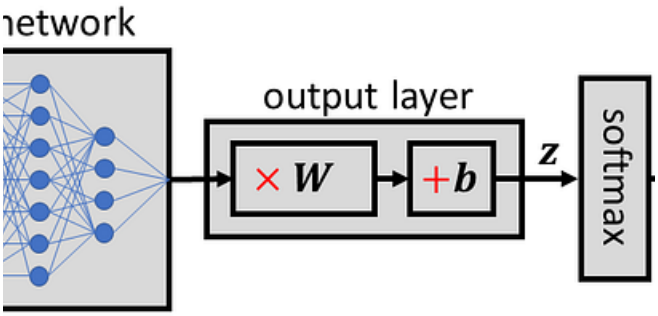
Written by Thomas Kurbiel

Following

336 Followers · Writer for Towards Data Science

Advanced Computer Vision & AI Research Engineer at APTIV Germany

More from Thomas Kurbiel and Towards Data Science



Thomas Kurbiel in Towards Data Science

Derivative of the Softmax Function and the Categorical Cross-Entrop...

A simple and quick derivation

6 min read · Apr 23, 2021

543 9

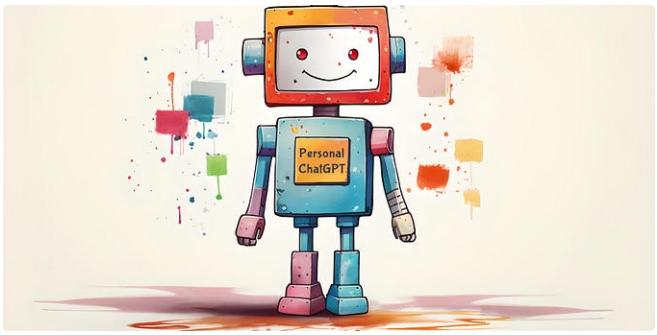
Antonis Makropoulos in Towards Data Science

How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and...

10 min read · Sep 17

549 11



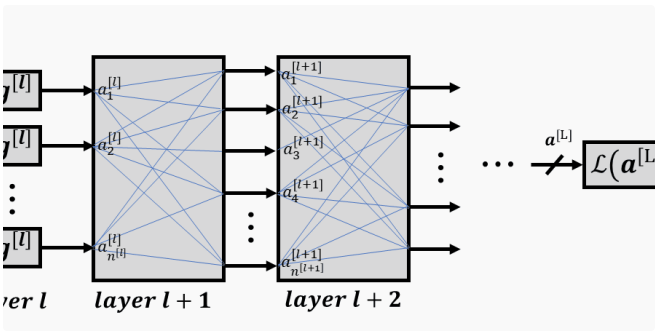
Robert A. Gonsalves in Towards Data Science

Your Own Personal ChatGPT

How you can fine-tune OpenAI's GPT-3.5 Turbo model to perform new tasks using you...

· 15 min read · Sep 8

595 7



Thomas Kurbiel in Towards Data Science

Deriving the Backpropagation Equations from Scratch (Part 1)

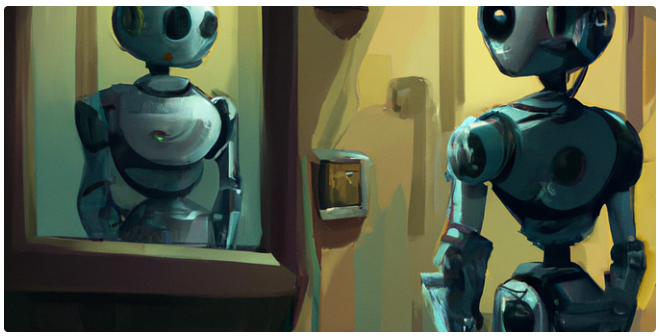
Gaining more insight into how neural networks are trained

7 min read · Nov 9, 2020

183 2

See all from Thomas Kurbiel | See all from Towards Data Science

Recommended from Medium



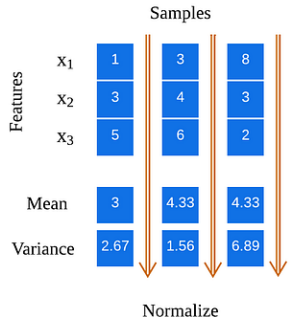
Thomas van Dongen in Towards Data Science

Demystifying efficient self-attention

A practical overview

20 min read · Nov 7, 2022

477 2



Ryan Partridge

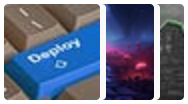
Inspecting Layer Normalization In Transformers

A Simple Trick For Improving Model Performance

✦ · 6 min read · Jul 1

12

Lists



Predictive Modeling w/ Python

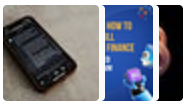
20 stories · 452 saves

Practical Guides to Machine Learning

10 stories · 519 saves

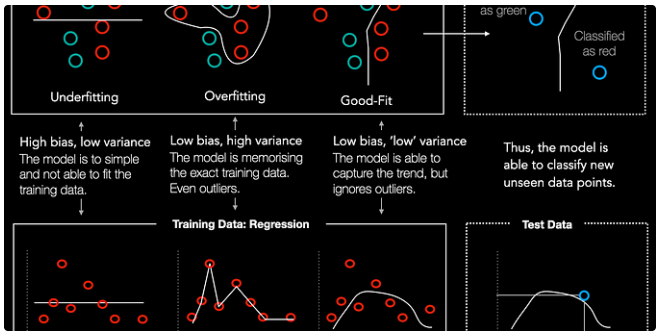
Natural Language Processing

669 stories · 283 saves



ChatGPT prompts

24 stories · 459 saves



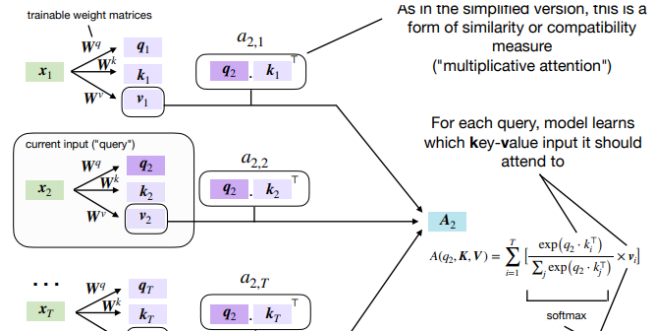
Frederik vl in Advanced Deep Learning

Understanding Bias and Variance in Machine Learning

The terms bias and variance describe how well the model fits the actual unknown data...

3 min read · Sep 15

44



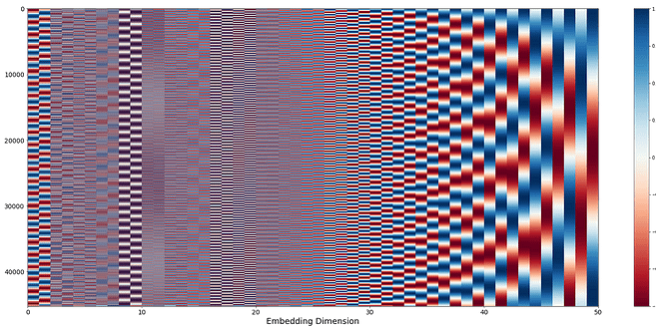
Zain ul Abideen

Attention Is All You Need: The Core Idea of the Transformer

An overview of the Transformer model and its key components.

6 min read · Jun 26

144

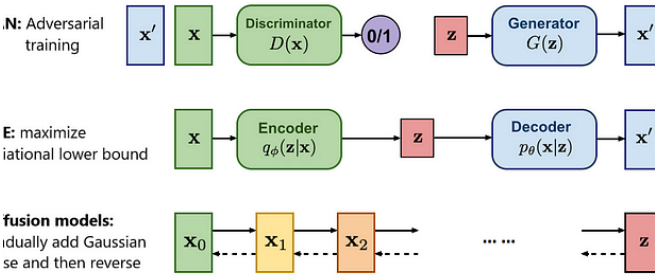


Eugene Ku

Transformer Architecture (Part 1—Positional Encoding)

Nowadays, arguably the most popular and influential model behind the hypes of deep...

4 min read · Aug 22



Ainur Gainetdinov in Towards AI

Diffusion Models vs GANs vs VAEs: Comparison of Deep Generative...

Deep generative models are applied to diverse domains such as image, audio, video...

6 min read · May 12


See more recommendations