A short introduction into deep learning ... and when to learn (and when not)

Christian Wolf

GDR IA²
October 28th, 2019
Scene understanding

Residual Conv-Deconv Grid Network for Semantic Segmentation
Damien Fourure, Rémi Emonet, Elisa Fromont, Damien Muselet, Alain Tremeau & Christian Wolf

[Fourure, Emonet, Fromont, Muselet, Tremeau, Wolf, BMVC 2017]
Vision + language

[Karpathy et al, 2015]
Learning to control

[Tan, Zhang, Coumans, Iscen, Bai, Hafner, Bohez, Vanhouke, RSS 2018]
1. Deep Learning

2. Feature Transfer and Domain Adaptation

3. To model or not to model?
Decision making

\[ \{\text{dog, cat, bike, avocado, blender, …}\} \]

\[ \{0, 1, \ldots, 26, 27, 28, \ldots, 98, 99, \ldots\} \]

\[ \{\text{Left, right, ahead, back, pick up, …}\} \]

Motor commands

“A blue parrot with a yellow belly sitting on a branch in a forest”
We would like to learn to predict a value $y$ from observed input $x$.

$$y = h(x, \theta)$$

Handcrafted from domain knowledge

Learned from data or interactions

Fully handcrafted

Fully Learned
Fitting and Generalisation

- Data are generated with function \( t = \sin(2\pi x) \)
- Objective: assuming the function unknown, predict \( t \) from \( x \)
Fitting and Generalisation

Example: « Fitting » of a polynomial of order $M$

$$y(x, w) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

« Least squares » (of errors) criterion

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, w) - t_n\}^2$$

Linear derivative -> direct solution
Model selection

Which order $M$ for the polynomial?

![Plot of polynomials having various orders $M$, shown as red curves, fitted to the data set shown in Figure 1.2.](image)

$E_{\text{RMS}} = \sqrt{\frac{2}{N} E(w^\star)}$ (1.3)

in which the division by $N$ allows us to compare different sizes of data sets on an equal footing, and the square root ensures that $E_{\text{RMS}}$ is measured on the same scale (and in the same units) as the target variable $t$.

Graphs of the training and test set RMS errors are shown, for various values of $M$, in Figure 1.5. The test set error is a measure of how well we are doing in predicting the values of $t$ for new data observations of $x$.

We note from Figure 1.5 that small values of $M$ give relatively large values of the test set error, and this can be attributed to the fact that the corresponding polynomials are rather inflexible and are incapable of capturing the oscillations in the function $\sin(2\pi x)$.

Values of $M$ in the range $3 \leq M \leq 8$ give small values for the test set error, and these also give reasonable representations of the generating function $\sin(2\pi x)$, as can be seen, for the case of $M = 3$, from Figure 1.4.

[C. Bishop, Pattern recognition and Machine learning, 2006]
Model selection

Separation into (at least) two sets
- Training set
- Validation set (hold out set)

Root Mean Square Error (RMS)

\[ E_{\text{RMS}} = \sqrt{2E(w^*)/N} \]
Cross validation

The separation changes iteratively. The validation error is the mean error over different « folds ».

[C. Bishop, Pattern recognition and Machine learning, 2006]
Big Data!

Overfitting increases if we increase the size of the training set.

\[ M = 9 \]
Regularisation

Add additional terms to the loss function which restrict expressivity of the model:

$$\tilde{E}(w) = \frac{1}{2} \sum_{n=1}^{N} \left\{ y(x_n, w) - t_n \right\}^2 + \frac{\lambda}{2} \| w \|^2$$

\[ \text{Regularisation parameter} \]

\[ w_0 \text{ is often excluded} \]

\[ M=9 \]

[C. Bishop, Pattern recognition and Machine learning, 2006]
The 3 fundamental problems of ML

1. Expressivity
   - What is the complexity of functions my model can represent?

2. Trainability
   - How easy is it to fit my model to my training data?

3. Generalization
   - Does my model generalize to unseen data?
   - In presence of shifts in distributions?

(After Eric Jang & Jascha Sohl-Dickstein)
Deep neural networks

Input layer

Hidden layer

Output layer

$w^{(1)}$

$w^{(2)}$

$L$
Going deeper

2012 : AlexNet, 8 layers. New techniques: dropout, ReLU


2015 : Microsoft research, 150 layers. New technique: residual learning
Gradient descent

One optimizer step:

$$\theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L} (h(x, \theta), y^*)$$

The gradient is a vector of partial derivatives:

$$\nabla \mathcal{L} = \begin{bmatrix}
\frac{\partial \mathcal{L}}{\partial \theta_0} \\
\frac{\partial \mathcal{L}}{\partial \theta_1} \\
\vdots \\
\frac{\partial \mathcal{L}}{\partial \theta_N}
\end{bmatrix}$$
Solution with finite differences

Before the publication of the gradient backpropagation algorithm, gradients were calculated by finite differences:

\[
\frac{\partial L}{\partial \theta_0} = \frac{L(x, \theta) - L(x, \theta + \Delta)}{L(x, \theta)} + O(\Delta)
\]

\[
\approx \frac{L(x, \theta) - L(x, \theta + \Delta)}{L(x, \theta)}
\]

→ approximate solution, high complexity.
Let's consider a linear network

\[ y_k = \sum_i w_{ki} x_i \]

and a sum of squared differences error function:

\[ \mathcal{L}_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \]

where samples are indexed by \( n \). The gradient w.r.t. sample \( n \) is:

\[ \frac{\partial \mathcal{L}}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni} \]
Differentiate a multi-layer network (1)

We need to calculate the derivative of a function which is a composition of several other functions, e.g. linear functions and point-wise non-linearities.

A two layer network can be written in the following form (omitting parameters in the notation):

\[ y = f_3(f_2(f_1(x))) \]

where

\[ f_1(x) = W^{(1)}x \]
\[ f_2(x) = \tanh(x) \]
\[ f_3(x) = W^{(2)}x \]
It’s all about chain rule of calculus...

Recall chain rule: given a function

\[ y(x) = f(g(x)) \]

or, in a slightly different notation, \( y = f \circ g \),

\[ y'(x) = f'(g(x))g'(x) \quad \text{(Lagrange’s notation)} \]

or

\[ (f \circ g)' = (f' \circ g) \cdot g' \]

or, if we name the intermediate variable \( z = g(x) \)

\[ \frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx} \quad \text{(Leibnitz’s notation)} \]

All notations are equivalent.
Differentiate a multi-layer network (2)

Let’s now consider a multi-layer network, in particular an arbitrary unit indexed by $j$ and receiving inputs from units indexed by $i$, providing outputs $z_i$. It’s activation $a_j$ (before the non-linearity) and output $z_j$ are:

$$a_j = \sum_i w_{ji} z_i, \quad z_j = h(a_j)$$

Its gradient is (using chain rule):

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

[C. Bishop, Pattern Recognition and Machine Learning, 2006]
Differentiate a multi-layer network (3)

\[
\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}
\]

Since \( a_j = \sum_i w_{ji}z_i \), we get

\[
\frac{\partial a_j}{\partial w_{ji}} = z_i
\]

We write / define:

\[
\delta_j \equiv \frac{\partial L}{\partial a_j}
\]

We obtain

\[
\frac{\partial L}{\partial w_{ji}} = \delta_j z_i
\]

⇒ we need to calculate \( \delta_j \) for each unit \( j \).

[C. Bishop, Pattern Recognition and Machine Learning, 2006]
Differentiate a multi-layer network (4)

If the output is linear (no activation) and we have a sum of squared differences loss, we get:

$$\delta_k = y_k - t_k$$

For the hidden units, we apply the chain rule:

$$\delta_j \equiv \frac{\partial L}{\partial a_j} = \sum_k \frac{\partial L}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

($$a_k$$ are units to which unit $$j$$ sends information.)

Rewriting, and taking into account the activation function $$h()$$, we get:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

$$\Rightarrow \delta_j$$ is calculated from the $$\delta_k$$.  
$$\Rightarrow$$ we traverse the network backwards!

[C. Bishop, Pattern Recognition and Machine Learning, 2006]
The full backpropagation algorithm

1. **Forward pass**: stimulate the model with input $x$, calculate all $a_j$ and $z_j$ up to the output $y$.

2. Calculate the $\delta_j$ for the output units using the derivative of the loss function.

3. **Backward pass**: calculate all $\delta_j$ with

\[
\delta_j = h'(a_j) \sum_k w_{kj} \delta_k
\]

4. Calculate the gradients with

\[
\frac{\partial L}{\partial w_{ji}} = \delta_j z_i
\]

[C. Bishop, Pattern Recognition and Machine Learning, 2006]
Remarks

- For batches, gradients are summed.
- The algorithm is general and can easily be adapted to other layers than fully-connected linear layers.
- The same algorithm can be used to calculate other gradients, e.g. deriving outputs with respect to inputs.
- Deep Learning frameworks calculate the gradients automatically given a definition of the forward pass (provided that the derivative of each sub function is available).
Backpropagation in PyTorch
Autograd

In PyTorch (and some other frameworks), Autograd performs automatic differentiation through a sequence of tensor instructions of an imperative language.

Let’s consider a simple linear operation:

\[ w = \begin{bmatrix} 5 & 3 \end{bmatrix}, \quad x = \begin{bmatrix} 7 & 2 \end{bmatrix}, \quad y = wx^T \]

The gradient of \( y \) w.r.t to \( x \) is given as

\[ \nabla = \begin{bmatrix} \frac{\partial y}{\partial x_i} \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix} \]

The gradient of \( y \) w.r.t to \( w \) is given as

\[ \nabla = \begin{bmatrix} \frac{\partial y}{\partial w_i} \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \end{bmatrix} \]
In PyTorch, we will first create the tensors:

```python
w = torch.tensor([5, 3], dtype=float, requires_grad=True)
x = torch.tensor([7, 1], dtype=float, requires_grad=True)
```

The `requires_grad` flag ensures that all calculations are tracked. We perform the linear operation:

```python
y = torch.dot(w, x)
```

Since the tensor $y$ has been calculated as result of operations on tracked tensors, it has a gradient function:

```python
print(y)
```

```
tensor(38., dtype=torch.float64, grad_fn=<DotBackward>)
```
We now run a backward pass on the variable $y$, which calculates gradients w.r.t. to all involved tensors:

```python
y.backward()
```

The gradients are attached to each variable:

```python
print (x.grad)
print (w.grad)
```

```
tensor([5., 3.], dtype=torch.float64)
tensor([7., 1.], dtype=torch.float64)
```
Autograd of a neural network

Recall our multi-layer network:

```python
class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = torch.nn.Linear(28*28, 300)
        self.fc2 = torch.nn.Linear(300, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

Here, `torch.nn.Linear(A, B)` sets up a $A \times B$ weight matrix and a bias vector. All backward passes will calculate gradients with respect to these tensors:

```python
model = MLP()
y = model(data)
loss = crossentropy(y, labels)
loss.backward()
```
1 Deep Learning

2 Feature Transfer and Domain Adaptation

3 To model or not to model?
IMAGENET Large Scale Visual Recognition Challenge

- 1,461,406 images dans ILSVRC 2010
- 1000 classes
- Annotation manuelle et vérification
- Crowd-sourcing (Amazon Mechanical Turk)

Fig. 4 Random selection of images in ILSVRC detection validation set. The images in the top 4 rows were taken from ILSVRC2012 single-object localization validation set, and the images in the bottom 4 rows were collected from Flickr using scene-level queries.

The second source (24%) is negative images which were part of the original ImageNet collection process but voted as negative: for example, some of the images were collected from Flickr and search engines for the ImageNet synset "animals" but during the manual verification step did not collect enough votes to be considered as containing an "animal." These images were manually re-verified for the detection task to ensure that they did not in fact contain the target objects. The third source (13%) is images collected from Flickr specifically for the detection task. These images were added for ILSVRC2014 following the same protocol as the second type of images in the validation and test set. This was done to bring the training and testing distributions closer together.
Visualizing feature map activations

For a given feature map, take the highest activations, and project them back to the image space to find out from which kind of signal they have been created.

[Zeiler and Fergus, ECCV 2014]
Fig. 2. Visualization of features in a fully trained model. For layers 2-5 we show the top activations as a random subset of features maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are not samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of discriminative parts of the image, e.g. eyes and noses of dogs (layer 4, row 1, cols 1). Best viewed in electronic form. The compression artifacts are a consequence of the 30Mb submission limit, not the reconstruction algorithm itself.

[Zeiler and Fergus, ECCV 2014]
Visualisation

Layer 3

[Zeiler and Fergus, ECCV 2014]
Visualisation

Layer 4

Layer 5

[Zeiler and Fergus, ECCV 2014]
How transferable are features in deep neural networks?

[Yosinski, Clune, Bengio, Lipson, ICML 2014]
Numerical/optimization issues (Positive or negative)

[Yosinski, Clune, Bengio, Lipson, "How transferable are features in deep neural networks?", 2014]
Where are my features?

[Ganin and Lempitsky, ICML 2015]
Combining real and simulated data

Joint positions (NYU Dataset)        Synthetic data (part segmentation)

Work of Natalia Neverova
Phd @ LIRIS, Now at Facebook AI

With Graham W. Taylor,
University of Guelph, Canada

Florian Nebout

[Neverova, Wolf, Taylor, Nebout. CVIU 2017]
Intermediate representation

We leverage an intermediate representation with strong topological properties to transfer from sim to real.

[Neverova, Wolf, Taylor, Nebout. CVIU 2017]
[Neverova, Wolf, Taylor, Nebout. CVIU 2017]
Learning dexterity and grasping

https://openai.com/blog/solving-rubiks-cube/

[Open-AI et al., October 2019]
Sim2real transfer

How can we transfer knowledge (eg policies) from simulations to real physical environments / robots?
Domain randomizations!

[Open-AI et al., October 2019]
Domain randomizations

[Open-AI et al., October 2019]
Domain Randomizations

- Simulator physics
- Generique noise
- Custom randomization:
  - Cube and robot friction
  - Cube size
  - Joint and tendon limits, margins
  - Action delay, latency, noise, Motor backlash
  - Gravity
- Vision:
  - Camera position, Rotation, field of view
  - Lighting conditions: rig, intensity
  - Materials
  - Color post processing

[Open-AI et al., October 2019]
Robustness to unseen perturbations

Unperturbed (for reference)

Rubber glove

Tied fingers

Blanket occlusion and perturbation

Plush giraffe perturbation

[Open-AI et al., October 2019]
1 Deep Learning

2 Feature Transfer and Domain Adaptation

3 To model or not to model?
\( \dot{x} \rightarrow \text{(Physical) model} \rightarrow y \)

\[
\ddot{\theta}_t = \frac{g \sin \theta_t + \cos \theta_t \left[ -F_t - ml \dot{\theta}_t^2 \sin \theta_t + \mu_c \text{sgn} (\dot{x}_t) \right]}{m_c + m} - \frac{\mu_p \dot{\theta}_t}{ml} \\
\dot{x}_t = \frac{F_t + ml \left[ \dot{\theta}_t^2 \sin \theta_t - \dot{\theta}_t \cos \theta_t \right] - \mu_c \text{sgn} (\dot{x}_t)}{m_c + m}
\]

Planning/shortest path
- Dijkstra
- A*
- Front Propagation
Shortest path problems: Dijkstra’s algorithm
Shortest path problems: Dijkstra’s algorithm
Shortest path problems: Dijkstra’s algorithm
Shortest path problems: Dijkstra’s algorithm
Shortest path problems: Dijkstra’s algorithm

Update of the bound
Shortest path problems: Dijkstra’s algorithm
Shortest path problems: Dijkstra’s algorithm
\[
\begin{align*}
\dot{\theta}_t &= \frac{g \sin \theta_t + \cos \theta_t \left[ -F_t - ml \dot{\theta}_t^2 \sin \theta_t + \mu_c \text{sgn} (\dot{x}_t) \right]}{m_c + m} - \frac{\mu_p \dot{\theta}_t}{ml} - l \left[ \frac{4}{3} - \frac{m \cos^2 \theta_t}{m_c + m} \right] \\
\dot{x}_t &= \frac{F_t + ml \left[ \dot{\theta}_t^2 \sin \theta_t - \dot{\theta}_t \cos \theta_t \right] - \mu_c \text{sgn} (\dot{x}_t)}{m_c + m}
\end{align*}
\]

Plannification/shortest path
- Dijkstra
- A*
- Front Propagation
- Physical model
- Black box (unknown function)
- White box (learned but known complex function)
- Hybrid model
Why we need inductive bias

The shortcomings of fully connected layers:

400x400 pixels

Flattened into 160,000 values

500 units

500 * 160,000 = 80 Million parameters (for a single layer!)
Parameters learned for one part of the image do not generalize to other parts of the image.
Shift-invariant linear operations

Classical MLPs are sequences of linear layers followed by point-wise non-linearities. What kind of linear and shift-invariant operators can there exist?

Let's consider an operator $\mathcal{O}$ with following properties:

**Linearity:**

$$\phi (\alpha f + \beta f') = \alpha \phi (f) + \beta \phi (f')$$

**Shift-invariance:**

$$\phi (m,n S(f)) = m,n S(\phi(f))$$

where $m,n S(f)$ shifts a signal by $m,n$.

**Impulse response** $h$:

$$h = \phi (0,0 p)$$

where $0,0 p$ is a Dirac impulse centered at 0, 0.
Shift-invariant linear operations

We decompose the signal $f$ into a series of Diracs $m,n$: 

$$[\phi(f)](x, y) = \left[ \phi \left( \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} f(m, n)^{m,n} \right) \right] (x, y)$$

We use linearity:

$$= \left[ \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} f(m, n) \phi^{m,n} \right] (x, y)$$

We use shift-invariance:

$$= \left[ \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} f(m, n) \phi^{m,n} S (0,0)^{m,n} \right] (x, y)$$

We use linearity again:

$$= \left[ \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} f(m, n)^{m,n} S (\phi^{0,0}) \right] (x, y)$$
Shift-invariant linear operations

Change in notation: we replace $\phi^{(0,0)p}$ by $h$ (through definition):

$$[\phi(f)](x, y) = \left[ \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} f(m, n)^{m,n} S(h) \right] (x, y)$$

Change in notation: make the shift-operator explicit:

$$= \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} f(m, n) h(x - m, y - n)$$

Change of variables:

$$= \sum_{m'=-M/2}^{M/2} \sum_{n'=-N/2}^{N/2} f(x - m', y - n') h(m', n')$$

$(m' = x - m, n' = y - n) \Rightarrow$ we get a convolution!
MLP vs LeNet on MNIST

Blue: training, Red: validation
Question:

What kind of inductive bias can we create for a given application?
Learning to navigate in 3D environments
Purely geometrical mapping + planning

Adding semantics

+ Real world solutions
- Simple tasks and reasoning (e.g. waypoint navigation)

Purely learned policies (Deep-RL)

Inductive biases: Geometry, topology, stability etc.

+ High-level reasoning
+ Discover tasks from reward
- Does not transfer to the real world
Active Neural Mapping

Deep networks trained with RL

[Chapelot, Gupta, Gandhi, Gupta, Salakhutdinov, 2019 (unpublished)]

Classical planning: shortest path, not trainable
Active Neural Mapping

[Chapelot, Gupta, Gandhi, Gupta, Salakhutdinov, 2019 (unpublished)]
Cognitive Mapping and Planning

- Differentiable planner (value iteration networks)
- End to end training, but no RL (imitation learning)
Purely geometrical mapping + planning

Adding semantics

Adding semantics

+ Real world solutions
- Simple tasks and reasoning (e.g. waypoint navigation)

Purely learned policies (Deep-RL)

Inductive biases:
Geometry, topology, stability etc.

+ High-level reasoning
+ Discover tasks from reward
- Does not transfer to the real world
Agent
Reasoning

Behavior / Control

Learning with different losses: Reward, (self)-supervision, Intrinsic motivation, curiosity

Memory
Semantic + spatial representation

Constraints + Structure:
Geometry, Topology, Semantics, Stability

Learning, querying, attending

Observations

π ν
Projective mapping
6 item scenario: time-step 005

Projective mapping of blue object

EgoMap: 3 Largest principal components
Object features retained in map

EgoMap: Query position

EgoMap: Attention
6 item scenario: time-step 105

EgoMap: Attention

EgoMap: Query position

EgoMap: 3 Largest principal components

Object features retained in map

EgoMap: Attention
6 item scenario: time-step 108

EgoMap: Query position

EgoMap: Attention

Collection of object n-1 triggers attention to object n

EgoMap: 3 Largest principal components
6 item scenario: time-step 134

EgoMap: Attention

EgoMap: Query position

EgoMap: 3 Largest principal components

EgoMap: Attention
When the object is not occluded, the agent does not attend to it.
Results

[Graph showing the comparison of Baseline, EgoMap, and Neural Map across Environment Frames.]

Upper bound on optimal Policy

Baseline
EgoMap
Neural Map

[Parisotto and Salakhutdinov, 2018]

[Beeching, Dibangoye, Simonin, Wolf, Under review]
Machine Learning and Control
Control Theory

Reinforcement Learning
The beauty and the beast

[Kaufmann, Gehrig, Foehn, Ranftl, Dosovitskiy2, Koltun, Scaramuzza1, 2018]
The beauty and the beast

Vision / perception: DNNs for state estimation and obstacle / door detection.

[Kaufmann, Gehrig, Foehn, Ranftl, Dosovitskiy, Koltun, Scaramuzza, 2018]
The beauty and the beast

Control: classical model predictive control

\[
\min_u \int_{t_0}^{t_f} \left( \bar{x}_t^T(t)Q\bar{x}_t(t) + \bar{u}_t^T(t)R\bar{u}_t(t) \right) dt
\]

\[
\bar{x}(t) = x(t) - x_r(t) \quad \bar{u}(t) = u(t) - u_r(t)
\]

subject to \( r(x, u) = 0 \) \( h(x, u) \leq 0 \).

[Kaufmann, Gehrig, Foehn, Ranftl, Dosovitskiy, Koltun, Scaramuzza, 2018]
Conclusion

Problems to solve:
- Expressivity: model symmetries and invariances in your data
- Trainability
- Generalization

1960-2012: let's handcraft features
2012-2016: let's handcraft architectures (layers, units ...)
2016- ?: let's handcraft data-flow (geometry, causality, attention)
External references (1)


OpenAI and Ilge Akkaya and Marcin Andrychowicz and Maciek Chociej and Mateusz Litwin and Bob McGrew and Arthur Petron and Alex Paino and Matthias Plappert and Glenn Powell and Raphael Ribas and Jonas Schneider and Nikolas Tezak and Jerry Tworek and Peter Welinder and Lilian Weng and Qiming Yuan and Wojciech Zaremba and Lei Zhang. Solving Rubik's Cube with a Robot Hand, axiv 2019.


Devendra Singh Chaplot, Saurabh Gupta, Dhiraj Gandhi, Abhinav Gupta, Ruslan Salakhutdinov, Active Neural Mapping, 2019 (unpublished)


References (our work)

[Neverova et al., 2017]


[Beeching et al., Arxiv 2018]


[Beeching et al., Under review]