Learning to Communicate: Challenges in optimising deep stochastic networks with categorical sampling

Jonathon Hare

Vision, Learning and Control
University of Southampton

The idea for this presentation comes from recent work with my PhD student Daniela Mihai in our upcoming NeurIPS workshop paper: Daniela Mihai and Jonathon Hare. "Avoiding hashing and encouraging visual semantics inreferential emergent language games". In: *3rd NeurIPS Workshop on Emergent Communication*. 2019. Some of the things I'll mention in passing relate to papers with another student, Yan Zhang, and Adam Prügel-Bennett.

Contents

Machine Learning - A Recap

All credit for this slide goes to Niranjan

Data	$\{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^N \qquad \{\boldsymbol{x}_n\}_{n=1}^N$
Function Approximator	$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{\theta}) + \nu$
Parameter Estimation	$E_0 = \sum_{n=1}^N \{\|\boldsymbol{y}_n - f(\boldsymbol{x}_n; \boldsymbol{\theta})\|\}^2$
Prediction	$\hat{\boldsymbol{y}}_{N+1} = f(\boldsymbol{x}_{N+1}, \hat{\boldsymbol{\theta}})$
Regularisation	$E_1 = \sum_{n=1}^N \{\|\boldsymbol{y}_n - f(\boldsymbol{x}_n; \boldsymbol{\theta})\|\}^2 + r(\|\boldsymbol{\theta}\|)$
Modelling Uncertainty	$p(\boldsymbol{\theta}\|\{\boldsymbol{x}_n, \boldsymbol{y}_n\}_{n=1}^N)$
Probabilistic Inference	$\mathbb{E}[g(\boldsymbol{\theta})] = \int g(\boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta} = \frac{1}{N_s} \sum_{n=1}^{N_s} g(\boldsymbol{\theta}^{(n)})$
Sequence Modelling	$\boldsymbol{x}_n = f(\boldsymbol{x}_{n-1}, \boldsymbol{\theta})$

What is Deep Learning?

Deep learning is primarily characterised by large datasets and function compositions with many parameters:

What is Deep Learning?

Deep learning is primarily characterised by large datasets and function compositions with many parameters:

- Feedforward networks: $\boldsymbol{y} = f(g(\boldsymbol{x}, \boldsymbol{\theta}_g), \boldsymbol{\theta}_f)$
  - Often with relatively simple functions - e.g.: $f(\boldsymbol{x}, \boldsymbol{\theta}_f) = \sigma(\boldsymbol{x}^\top \boldsymbol{\theta}_f)$ or $f(\boldsymbol{x}, \boldsymbol{\theta}_f) = \sigma(\boldsymbol{x} \star \boldsymbol{\theta}_f)$

What is Deep Learning?

Deep learning is primarily characterised by large datasets and function compositions with many parameters:

- Feedforward networks: $\boldsymbol{y} = f(g(\boldsymbol{x}, \boldsymbol{\theta}_g), \boldsymbol{\theta}_f)$
  - Often with relatively simple functions - e.g.: $f(\boldsymbol{x}, \boldsymbol{\theta}_f) = \sigma(\boldsymbol{x}^\top \boldsymbol{\theta}_f)$ or $f(\boldsymbol{x}, \boldsymbol{\theta}_f) = \sigma(\boldsymbol{x} \star \boldsymbol{\theta}_f)$
- Recurrent networks: $\boldsymbol{y}_t = f(\boldsymbol{y}_{t-1}, \boldsymbol{x}_t, \boldsymbol{\theta}) = f(f(\boldsymbol{y}_{t-2}, \boldsymbol{x}_{t-1}, \boldsymbol{\theta}), \boldsymbol{\theta}) = \ldots$

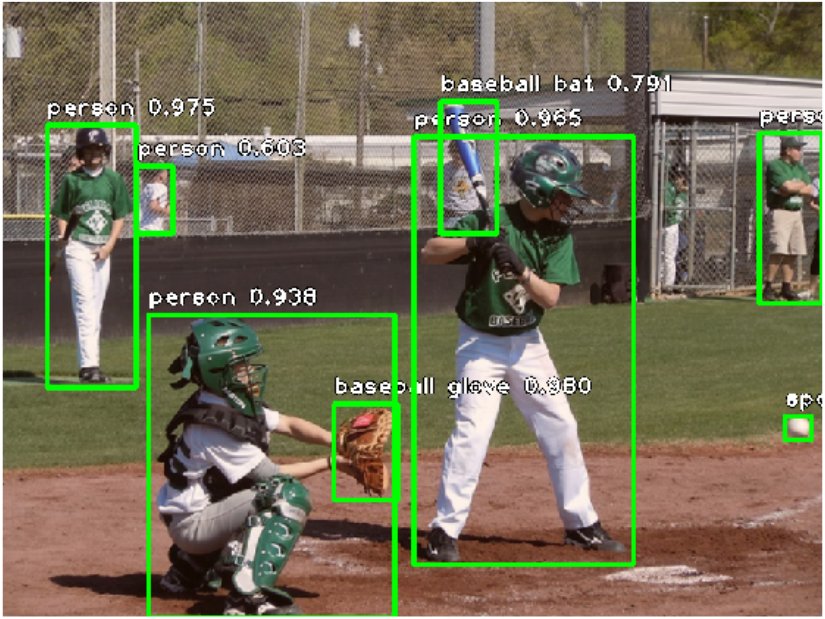
What is Deep Learning?

Deep learning is primarily characterised by large datasets and function compositions with many parameters:

- Feedforward networks: $\boldsymbol{y} = f(g(\boldsymbol{x}, \boldsymbol{\theta}_g), \boldsymbol{\theta}_f)$
  - Often with relatively simple functions - e.g.: $f(\boldsymbol{x}, \boldsymbol{\theta}_f) = \sigma(\boldsymbol{x}^\top \boldsymbol{\theta}_f)$ or $f(\boldsymbol{x}, \boldsymbol{\theta}_f) = \sigma(\boldsymbol{x} \star \boldsymbol{\theta}_f)$
- Recurrent networks: $\boldsymbol{y}_t = f(\boldsymbol{y}_{t-1}, \boldsymbol{x}_t, \boldsymbol{\theta}) = f(f(\boldsymbol{y}_{t-2}, \boldsymbol{x}_{t-1}, \boldsymbol{\theta}), \boldsymbol{\theta}) = \ldots$

In the early days the focus of deep learning was on learning functions for classification. Nowadays the functions are much more general in their inputs and outputs.

A deep learning example: object detection



Differentiable Programming

- Differentiable programming is a term coined by Yann Lecun[1] to describe a superset of Deep Learning.
- Captures the idea that computer programs can be constructed of parameterised functional blocks in which the parameters are learned using some form of **gradient-based optimisation.**
  - The implication is that we need to be able to compute gradients with respect to the parameters of these functional blocks.

[1] https://www.facebook.com/yann.lecun/posts/10155003011462143

[2] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. "Learning Representations of Sets through Optimized Permutations". In: *International Conference on Learning Representations*. 2019.

[3] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. "Deep Set Prediction Networks". In: *Advances in Neural Information Processing Systems 32*. 2019.

Differentiable Programming

- Differentiable programming is a term coined by Yann Lecun[1] to describe a superset of Deep Learning.
- Captures the idea that computer programs can be constructed of parameterised functional blocks in which the parameters are learned using some form of **gradient-based optimisation.**
  - The implication is that we need to be able to compute gradients with respect to the parameters of these functional blocks.
  - The idea of Differentiable Programming also opens up interesting possibilities:
    - The functional blocks don't need to be direct functions in a mathematical sense; more generally they can be *algorithms*.
    - What if the functional block we're learning parameters for is itself an algorithm that optimises the parameters of an internal algorithm using a gradient based optimiser?![2,3]

[1] https://www.facebook.com/yann.lecun/posts/10155003011462143

[2] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. "Learning Representations of Sets through Optimized Permutations". In: *International Conference on Learning Representations*. 2019.

[3] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. "Deep Set Prediction Networks". In: *Advances in Neural Information Processing Systems 32*. 2019.

Learning to Play Communication Games: Motivation

- As humans, our understanding and conceptualisation of the visual world is very much linked with our ability to communicate using natural language.
- Machine learners (and mathematicians) have long understood that forcing data through a bottleneck can lead to representations that capture information.
- **We want to explore if a 'language bottleneck' between a pair of agents performing collaborative tasks can encourage the emergence of a communication protocol that captures visual semantics.**
  - We'll assume a 'language bottleneck' means a variable length sequence of discrete tokens drawn from a fixed size vocabulary.

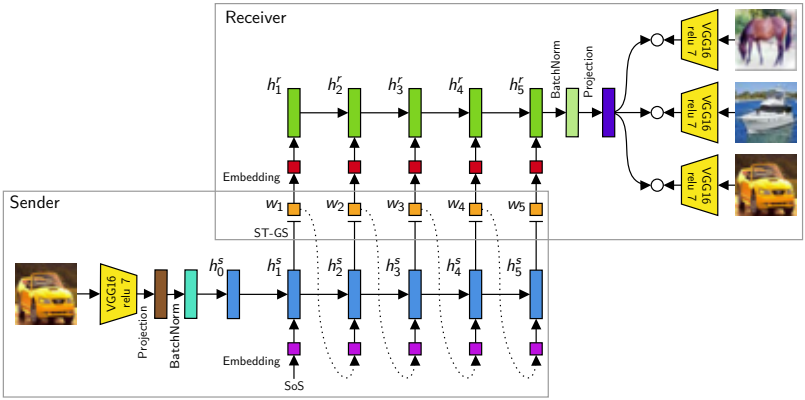
A Referential Communication Game

A Referential Game[4]: Alice must communicate to Bob which image she has (Bob has that image, plus many distractors). Communication is one-way only. Alice knows nothing about the distractors Bob has (they could all be white boats!).

[4] David K. Lewis. *Convention: A Philosophical Study*. Wiley-Blackwell, 1969.

A Computational Model

This is the model proposed by Havrylov and Titov[5]:



$$\mathcal{L}_{game;\phi,\theta}(t) = \mathbb{E}_{m_t \sim p_\phi(\cdot|t)} \left[\sum_{k=1}^{K} \max[0, 1 - f(t)^\top g(h_l^r) + f(d_k)^\top g(h_l^r)] \right]$$

[5] Serhii Havrylov and Ivan Titov. "Emergence of Language with Multi-agent Games: Learning to Communicate with Sequences of Symbols". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 2149–2159.

How can we optimise the parameters of this model? (I)

There are a few challenges:

- We have a non-differentiable sampling operation in the middle of the model (sampling tokens from a Categorical distribution).

[6] Admittedly many of these are wrapped up in vectors/matrices/tensors so the actual number of analytical expressions would be significantly less

How can we optimise the parameters of this model? (I)

There are a few challenges:

- We have a non-differentiable sampling operation in the middle of the model (sampling tokens from a Categorical distribution).
- This model is complex:
  - we could in principle hand-derive an analytical expression for the derivative of the loss with respect to each parameter, but...
  - the model is deep — you would need to apply the chain rule a huge number of times,
  - the depth is actually variable because the messages are variable length,
  - you'll be dealing with derivatives of sigmoid's, tanh's and ReLUs, and,
  - there are **6,171,493** learnable parameters[6]!!!

[6] Admittedly many of these are wrapped up in vectors/matrices/tensors so the actual number of analytical expressions would be significantly less

How can we optimise the parameters of this model? (II)

There are a few challenges:

- Even with expressions for the first-order derivatives, the optimisation problem is *hard* - in particular, the loss landscape is obviously high dimensional, but it also has:
  - many local minima, and
  - a considerable amount of symmetry.

How can we optimise the parameters of this model? (II)

There are a few challenges:

- Even with expressions for the first-order derivatives, the optimisation problem is *hard* - in particular, the loss landscape is obviously high dimensional, but it also has:
  - many local minima, and
  - a considerable amount of symmetry.
- There is significant computational complexity; there is no hope of performing full gradient descent:
  - With just 128 images (1 target & 127 distractors) of size 228x228, the memory requirements for storing the intermediate results (required for computing the gradients) exceeds 38GB.
  - The forward-pass takes of the order of 10^{10} multiply-accumulate CPU operations.

Differentiable Sampling: Straight-Through Gumbel Softmax

The generation of a discrete token, t, from a vocabulary of K tokens is achieved by sampling a categorical distribution

$$t \sim \text{Cat}(p_1, \ldots, p_K)\,; \sum_i p_i = 1.$$

Differentiable Sampling: Straight-Through Gumbel Softmax

The generation of a discrete token, t, from a vocabulary of K tokens is achieved by sampling a categorical distribution

$$t \sim \text{Cat}(p_1, \ldots, p_K)\,; \sum_i p_i = 1.$$

Generating the probabilities $p_1, \ldots, p_K$ directly from a neural network has potential numerical problems; it's much easier to generate un-normalised log-probabilities (logits), $x_1, \ldots, x_K$.

Differentiable Sampling: Straight-Through Gumbel Softmax

The generation of a discrete token, t, from a vocabulary of K tokens is achieved by sampling a categorical distribution

$$t \sim \text{Cat}(p_1, \ldots, p_K)\,; \sum_i p_i = 1.$$

Generating the probabilities $p_1, \ldots, p_K$ directly from a neural network has potential numerical problems; it's much easier to generate un-normalised log-probabilities (logits), $x_1, \ldots, x_K$.

The gumbel-softmax trick allows us to sample directly using the logits:

$$t = \underset{i \in \{1, \cdots, K\}}{\text{argmax}}\ x_i + z_i$$

where $z_1, \ldots z_K$ are i.i.d Gumbel(0,1) variates which can be computed from Uniform variates through $-\log(-\log(-\mathcal{U}(0,1)))$.

Differentiable Sampling: Straight-Through Gumbel Softmax

Ok, but how does that help? argmax isn't differentiable!

[7] in practice we'll either use an annealing schedule for the temperature, or just estimate a good value for a given $\boldsymbol{y}$ using an additional small network built into the model!

Differentiable Sampling: Straight-Through Gumbel Softmax

Ok, but how does that help? argmax isn't differentiable!

...softargmax is:

$$\text{softargmax}(\boldsymbol{y}) = \sum_i \frac{e^{\beta y_i}}{\sum_j e^{\beta y_j}} i$$

where β is the inverse-temperature parameter. Higher values of β give a more peaky distribution[7].

[7] in practice we'll either use an annealing schedule for the temperature, or just estimate a good value for a given $\boldsymbol{y}$ using an additional small network built into the model!

Differentiable Sampling: Straight-Through Gumbel Softmax

But... this clearly gives us a result that will be non-integer; we cannot round or clip because it would be non-differentiable.

Differentiable Sampling: Straight-Through Gumbel Softmax

But... this clearly gives us a result that will be non-integer; we cannot round or clip because it would be non-differentiable.

The Straight-Through operator allows us to take the result of a true argmax that has the gradient of the softargmax:

$$\text{STargmax}(\boldsymbol{y}) = \text{softargmax}(\boldsymbol{y}) + \text{stopgradient}(\text{argmax}(\boldsymbol{y}) - \text{softargmax}(\boldsymbol{y}))$$

where stopgradient is defined such that $\text{stopgradient}(\boldsymbol{a}) = \boldsymbol{a}$ and $\nabla \, \text{stopgradient}(\boldsymbol{a}) = 0$.

Differentiable Sampling: Straight-Through Gumbel Softmax

But... this clearly gives us a result that will be non-integer; we cannot round or clip because it would be non-differentiable.

The Straight-Through operator allows us to take the result of a true argmax that has the gradient of the softargmax:

$$\text{STargmax}(\boldsymbol{y}) = \text{softargmax}(\boldsymbol{y}) + \text{stopgradient}(\text{argmax}(\boldsymbol{y}) - \text{softargmax}(\boldsymbol{y}))$$

where stopgradient is defined such that $\text{stopgradient}(\boldsymbol{a}) = \boldsymbol{a}$ and $\nabla \text{stopgradient}(\boldsymbol{a}) = 0$.

Straight-Through Gumbel Softmax

Combine the gumbel softmax trick with the STargmax to give you discrete samples, with a useable gradient[a].

[a] The ST operator is biased but low variance; in practice it works very well and is better than the high-variance unbiased estimates you could get through REINFORCE.

How are we going to compute all the gradients needed for out optimiser?

To solve optimisation problems using gradient methods we need to compute the gradients (derivatives) of the objective with respect to the parameters.

In our model we're talking about the gradients of the hinge loss function, $\mathcal{L}$ with respect to the 6.2M parameters $\boldsymbol{\theta}$: $\nabla_{\boldsymbol{\theta}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$

Computing Derivatives

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

Computing Derivatives

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

Problems

Static - can't "differentiate algorithms"; unwieldy with millions of variables

Computing Derivatives

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

Problems

Numerical errors - will compound in deep nets

Computing Derivatives

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

What is Automatic Differentiation (AD)?

Automatic Differentiation is:

- a method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards.
  - Takes code that computes a function and uses that to compute the derivative of that function.
  - The goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

What is Automatic Differentiation (AD)?

Automatic Differentiation is:

- a method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards.
  - Takes code that computes a function and uses that to compute the derivative of that function.
  - The goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Reverse-Mode Automatic Differentiation

Modern dynamic differentiable programming libraries are built around a technique called Reverse Mode Automatic Differentiation.

You write the forward pass; when you run the program, a 'computation graph' can be built dynamically, and this graph contains the 'adjoint variables' which can be used to compute gradients by applying the chain rule.

Optimisation

- We need to limit outselves to first-order methods.
- Gradient Descent is conceptually simple.
- Our loss landscape is wild.

Computing the Hessian is not going to be feasible!

Optimisation

- We need to limit outselves to first-order methods.
- Gradient Descent is conceptually simple.
- Our loss landscape is wild.

Compute the gradient at the current point and take a step proportional to the gradient in the steepest direction.

Optimisation

- We need to limit outselves to first-order methods.
- Gradient Descent is conceptually simple.
- Our loss landscape is wild.

Utilise stochasticity to help explore (e.g. SGD).

Utilise adaptive learning rates computed from the history of past moves ('momentum') to 'ride over bumps' - algorithms such as ADAM.

Try to pick good starting points (e.g. He/Xavier(Glorot) initialisation).

Use batch normalisation in the model to keep weights (and their gradients) in check.

Computation

- Our model is computationally extreme, yet we still would like to experiment with it in a reasonable amount of time.
- GPUs which enabled the Deep Learning Revolution to start in 2012 are still just as relevant today.
- Our model requires 4 RTX2080ti GPUs (each with 11G RAM) to run!
  - The VGG16 backbone sits on one GPU
  - The Sender and Reciever are each on a different GPU
  - The loss computation takes place on the forth GPU
- For efficiency we use mini-batch SGD with ADAM to train
  - mini-batch size of 128
  - allows for 127 distractors and one target
  - but we can utilise all 128 possible targets in a single iteration
- 74504 images in the training set; but this is made essentially infinite through data augmentation.
- During training we can play and learn from 74496 games in about 7mins 30secs. It takes around 4 hours of play for convergence!

Computation

- Our model is computationally extreme, yet we still would like to experiment with it in a reasonable amount of time.
- GPUs which enabled the Deep Learning Revolution to start in 2012 are still just as relevant today.
- Our model requires 4 RTX2080ti GPUs (each with 11G RAM) to run!
  - The VGG16 backbone sits on one GPU
  - The Sender and Reciever are each on a different GPU
  - The loss computation takes place on the forth GPU
- For efficiency we use mini-batch SGD with ADAM to train
  - mini-batch size of 128
  - allows for 127 distractors and one target
  - but we can utilise all 128 possible targets in a single iteration
- 74504 images in the training set; but this is made essentially infinite through data augmentation.
- During training we can play and learn from 74496 games in about 7mins 30secs. It takes around 4 hours of play for convergence!

Computation

- Our model is computationally extreme, yet we still would like to experiment with it in a reasonable amount of time.
- GPUs which enabled the Deep Learning Revolution to start in 2012 are still just as relevant today.
- Our model requires 4 RTX2080ti GPUs (each with 11G RAM) to run!
  - The VGG16 backbone sits on one GPU
  - The Sender and Reciever are each on a different GPU
  - The loss computation takes place on the forth GPU
- For efficiency we use mini-batch SGD with ADAM to train
  - mini-batch size of 128
  - allows for 127 distractors and one target
  - but we can utilise all 128 possible targets in a single iteration
- 74504 images in the training set; but this is made essentially infinite through data augmentation.
- During training we can play and learn from 74496 games in about 7mins 30secs. It takes around 4 hours of play for convergence!

Experiments

In our NeuIPS EmeCom workshop paper[8] we explore Havrylov and Titov's model under a number of different configurations.

In all cases we shrunk the dataset to 32x32 images from the CIFAR-10 dataset, reduced the allowed vocabulary size to 100 tokens, and setting the maximum sentence length to 5.

- This makes the model much more manageable - you can *just* fit the whole thing on open GPU with 127 distractors + 1 target.

[8] Daniela Mihai and Jonathon Hare. "Avoiding hashing and encouraging visual semantics inreferential emergent language games". In: *3rd NeurIPS Workshop on Emergent Communication*. 2019.

Experiments

In our NeuIPS EmeCom workshop paper[8] we explore Havrylov and Titov's model under a number of different configurations.

In all cases we shrunk the dataset to 32x32 images from the CIFAR-10 dataset, reduced the allowed vocabulary size to 100 tokens, and setting the maximum sentence length to 5.

- This makes the model much more manageable - you can *just* fit the whole thing on open GPU with 127 distractors + 1 target.

Key metrics are:

Comm. success rate	proportion of fully successfully games
Top-5 comm. success rate	proportion of almost successful games
#target class in top-5	#images in the top five with target label
Target class avg. rank	average rank of images with the target label

[8] Daniela Mihai and Jonathon Hare. "Avoiding hashing and encouraging visual semantics inreferential emergent language games". In: *3rd NeurIPS Workshop on Emergent Communication*. 2019.

Experiments & Findings 1: feature extraction

We firstly explored the importance of the VGG16 feature extraction network, and explored the effect of

- learning it as part of the model;
- keeping it fixed with random weights; and
- keeping it fixed with pretrained weights from ImageNet (essentially 'importing' external semantic knowledge into the network).

Experiments & Findings 1: feature extraction

We firstly explored the importance of the VGG16 feature extraction network, and explored the effect of

- learning it as part of the model;
- keeping it fixed with random weights; and
- keeping it fixed with pretrained weights from ImageNet (essentially 'importing' external semantic knowledge into the network).

Feature extractor	Comm. rate	Top-5 comm. rate	#target-class in top-5	Target-class avg. rank
Pretrained & fixed	0.88 ($\pm$0.31)	0.99	1.84	46.58
Random & frozen	0.95 ($\pm$0.19)	1	1.66	52.13
Learned end-end	0.89 ($\pm$0.3)	1	1.53	54.01

Visual semantics are much stronger in the pretrained model, but they don't make for the best game-play. A fully learned model doesn't learn any notion of semantics.

Experiments & Findings 2: augmentation

We then explored the effect of using augmentations to make the game harder.

Experiments & Findings 2: augmentation

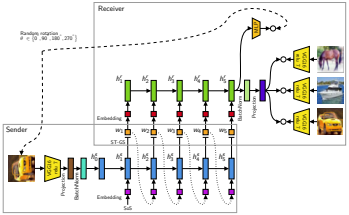
We then explored the effect of using augmentations to make the game harder.

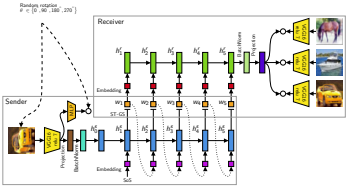
Feature extractor	Comm. rate	Top-5 comm. rate	#target-class in top-5	Target-class avg. rank
Sender images augmented with Gaussian noise:				
Pretrained & fixed	0.92 ($\pm$0.26)	0.99	1.85	46.12
Random & frozen	0.96 ($\pm$0.19)	1	1.6	54.01
Learned end-end	0.94 ($\pm$0.23)	1	1.5	57.11
Sender images augmented with random rotations:				
Pretrained & fixed	0.83 ($\pm$0.37)	0.99	2.07	41.89
Random & frozen	0.87 ($\pm$0.33)	0.99	1.7	51.89
Learned end-end	0.92 ($\pm$0.25)	1	1.6	55.96

Adding noise makes the communication rate increase; adding rotations makes the semantics increase (but decreases communication success rate).

Experiments & Findings 3: multiple tasks

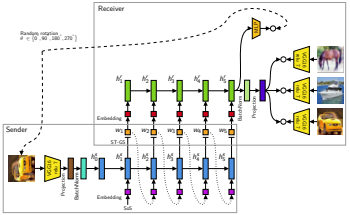
Finally we extended the game to include a secondary task (guessing the rotation of the sender's input) in order to assess whether having agents perform more diverse tasks might lead to stronger visual semantics emerging.

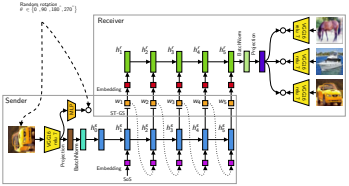




Experiments & Findings 3: multiple tasks

Finally we extended the game to include a secondary task (guessing the rotation of the sender's input) in order to assess whether having agents perform more diverse tasks might lead to stronger visual semantics emerging.





Model	Comm. rate	Top-5 comm. rate	#target-class in top-5	Target-class avg. rank	Rot. acc.
Receiver-Predicts (l)	0.64 ($\pm$0.48)	0.95	1.84	45.6	0.82
Sender-Predicts (r)	0.69 ($\pm$0.46)	0.98	2.05	43.41	0.84

Very good visual semantic capture (with only self-supervision); okay, but high variance game play.

Open questions

- Those last models are *really* difficult to optimise.
  - The losses of the different tasks pull in different directions.
  - For the second (sender predicts) model we optimised $0.5 \cdot \mathcal{L}_{rotation} + \mathcal{L}_{game}$, where $\mathcal{L}_{game}$
  - For the first (receiver predicts) model we switched between $5.0 \cdot \mathcal{L}_{rotation}$ and $5.0 \cdot \mathcal{L}_{rotation} + \mathcal{L}_{game}$ on alternate batch iterations.
- Can we fix this?
  - We tried an additive loss with learned weights[9] without success.
  - Multi-objective optimisation (e.g. using the Frank-Wolfe algorithm to find a direction to move which satisfies both loss terms[10])?
  - something else?

[9] Alex Kendall, Yarin Gal, and Roberto Cipolla. "Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

[10] Ozan Sener and Vladlen Koltun. "Multi-Task Learning as Multi-Objective Optimization". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 527–538.

Conclusions

- Modern deep learning/differentiable programming lets us build neat models.
- State-of-the-art first-order stochastic optimisation works surprisingly well, but it can be slow.
- To build models that get us closer to human-like abilities be need to be able to optimise losses for multiple tasks in the same framework whilst ensuring there is enough balance to allow the model to generalise to all the tasks.

Thank you!

Any questions?