

# MATH 6380P ADVANCED TOPICS IN DEEP LEARNING NEXPERIA CHALLENGE

Andrea Madotto, Genta Indra Winata,  
Zhaojiang Lin, Jamin Shin

CAiRE lab

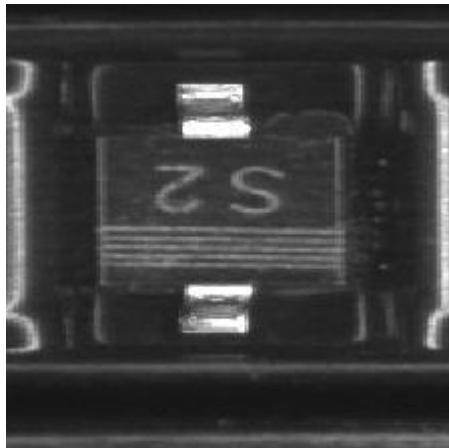
# TABLE OF CONTENTS

- Dataset and baselines
- Automatic Architectural Search
  - General Idea (ENAS)
  - DARTS
- Visualization (GAM)
- Q&A

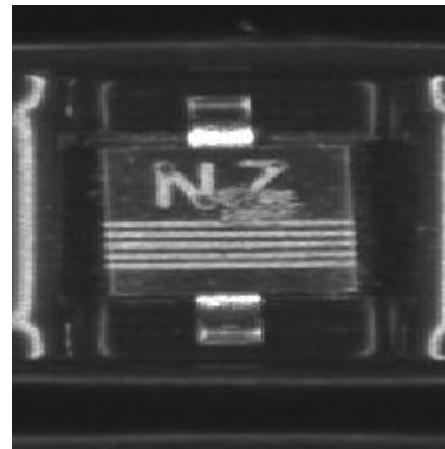
# DATASET: IMAGE CLASSIFICATION OF SEMI-CONDUCTOR

- 2 classes, **30K** training and **3K** testing images
- Validation split: 300 images (150 bad and 150 good).
- **No particular pre-processing** (already preprocessed)

GOOD



BAD



# BASELINE AND RESULTS

- **Feature Based**(Res-Net feature)
  - LDA
  - Random Forest (RM)
  - Logistic Regression (LR)
  - Support Vector Machine (SVM)
- **From Scratch**
  - CNN (2d conv, 2-layer)
- **Transferring Learning**
  - Res-Net LL (last-layer)
  - Res-Net FBP (full backpropagation)

Model	Valid Acc	Test AUC
<i>LDA</i>	58.0	-
<i>RF</i>	54.0	-
<i>LR</i>	55.3	-
<i>SVM</i>	57.0	-
<i>Res-Net LL</i>	60.67	-
<i>2-layer CNN</i>	77.0	-
<i>Res-Net FBP</i>	<b>97.33</b>	98.30

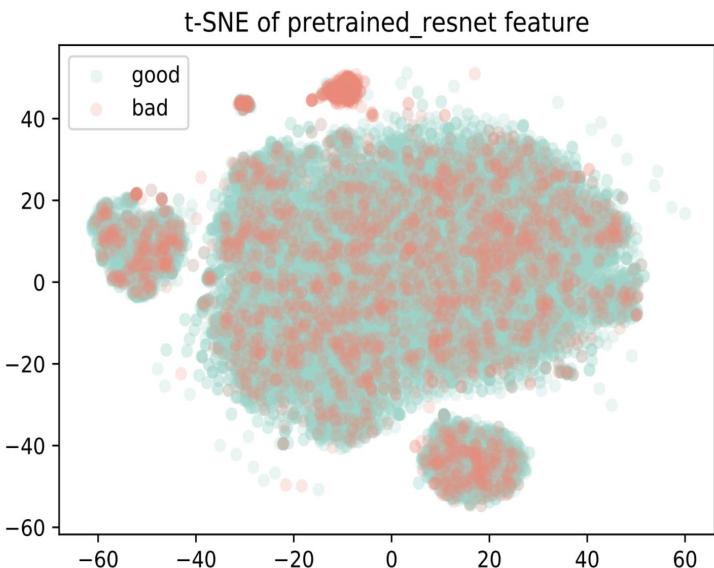
# BASELINE AND RESULTS

- **Feature Based**(Res-Net feature)
  - LDA
  - Random Forest (RM)
  - Logistic Regression (LR)
  - Support Vector Machine (SVM)
- **From Scratch**
  - CNN (2d conv, 2-layer)
- **Transferring Learning**
  - Res-Net LL (last-layer)
  - Res-Net FBP (full backpropagation)

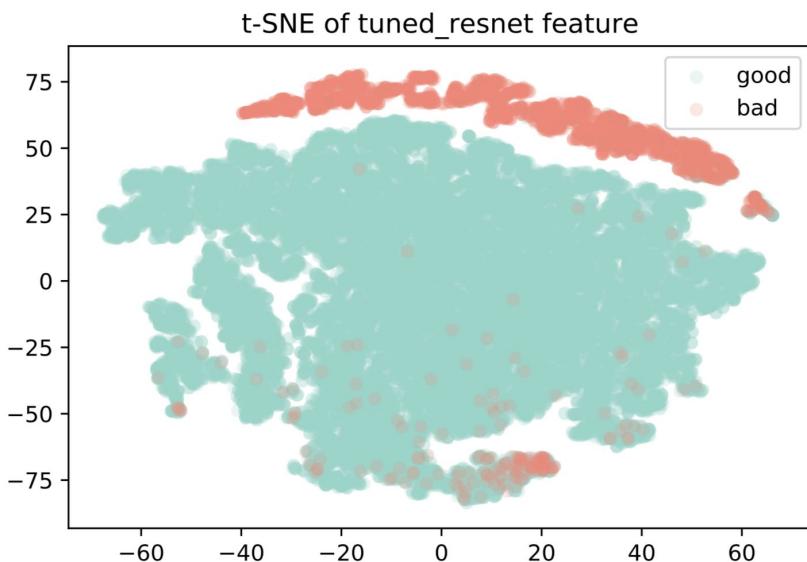
Res-Net FBP  
features

Model	Valid Acc	Test AUC
LDA	58.0 (96.0)	-
RF	54.0 (95.6)	-
LR	55.3 (95.6)	-
SVM	57.0 (95.6)	-
<i>Res-Net LL</i>	60.67	-
<i>2-layer CNN</i>	77.0	-
<i>Res-Net FBP</i>	<b>97.33</b>	98.30

# T-SNE VISUALIZATION



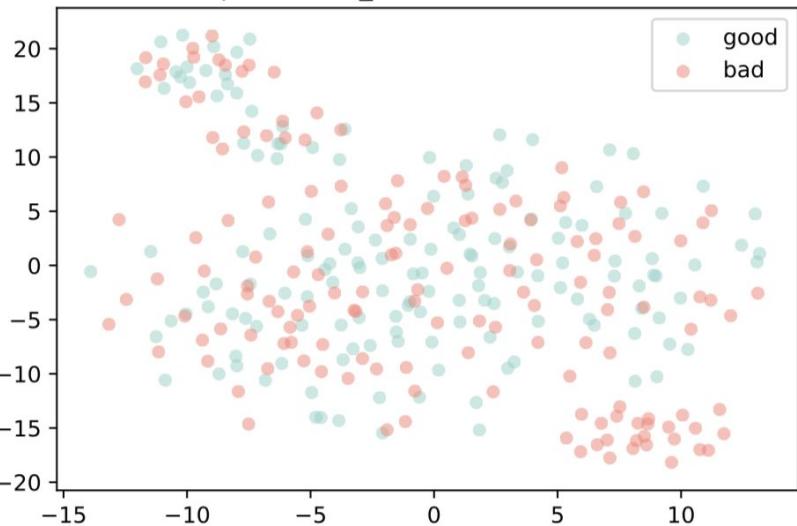
(a) Features of the pretrained Res-Net



(b) Features of fine-tuned Res-Net

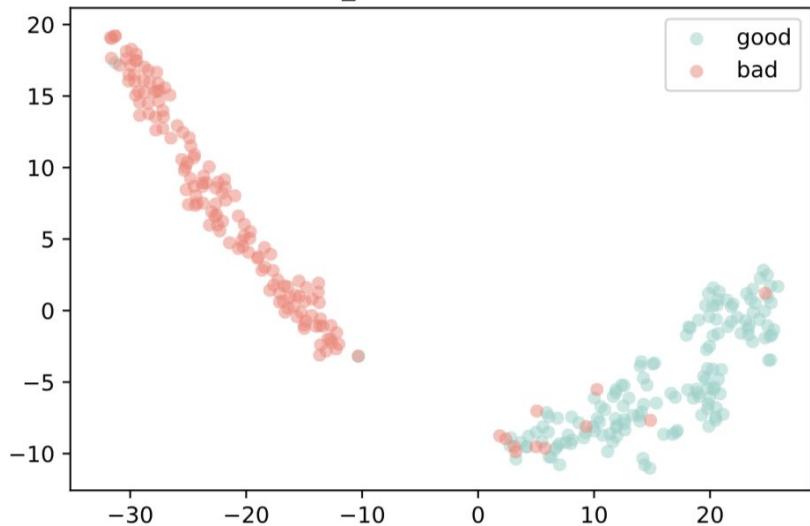
# T-SNE VISUALIZATION

t-SNE of pretrained\_resnet validation set feature



(a) Features of the pretrained Res-Net

t-SNE of tuned\_resnet validation set feature



(b) Features of fine-tuned Res-Net

# AUTOMATIC ARCHITECTURAL SEARCH

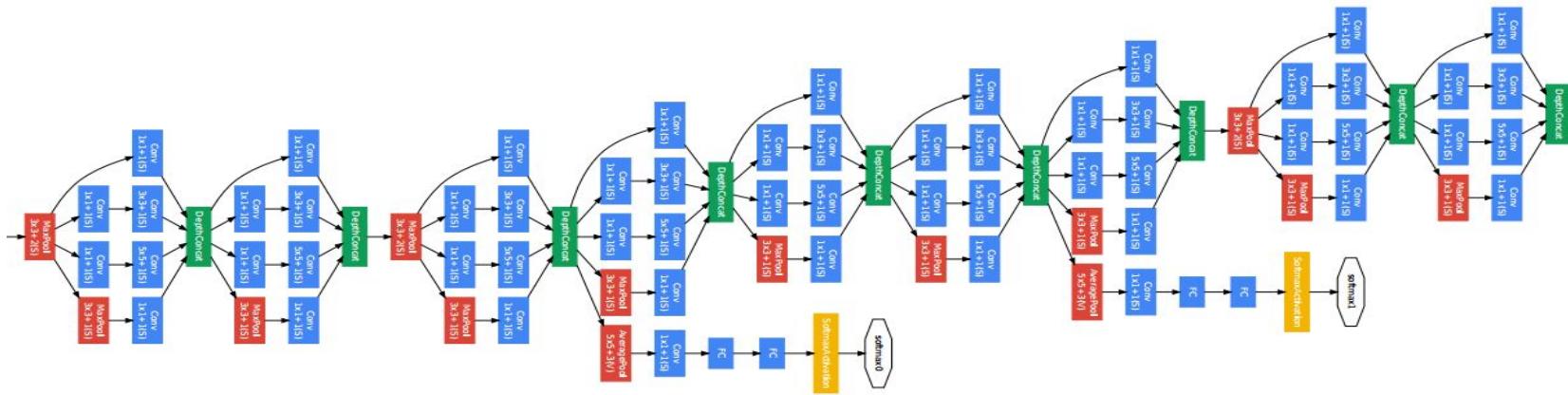
# WHAT'S MORE?

- Transferring Learning (Res-Net FBP) is already very good
- Learning from scratch (CNN) not so good instead\*\*
- The idea:
  - **Automatically search** an architecture that generalize well
  - Computationally expensive task:
    - Try different **hyper-parameters** (configuration) and select the one with highest validation set accuracy
    - **Optimize** (e.g. Reinforcement Learning) a function w.r.t. to configuration space
  - Several recent paper try to make it less computationally expensive:
    - Efficient Neural Architecture Search via Parameter Sharing (**ENAS**)
    - Differentiable ARchiTecture Search (**DARTS**)

\*\* We did not tune much, but likely to overfit

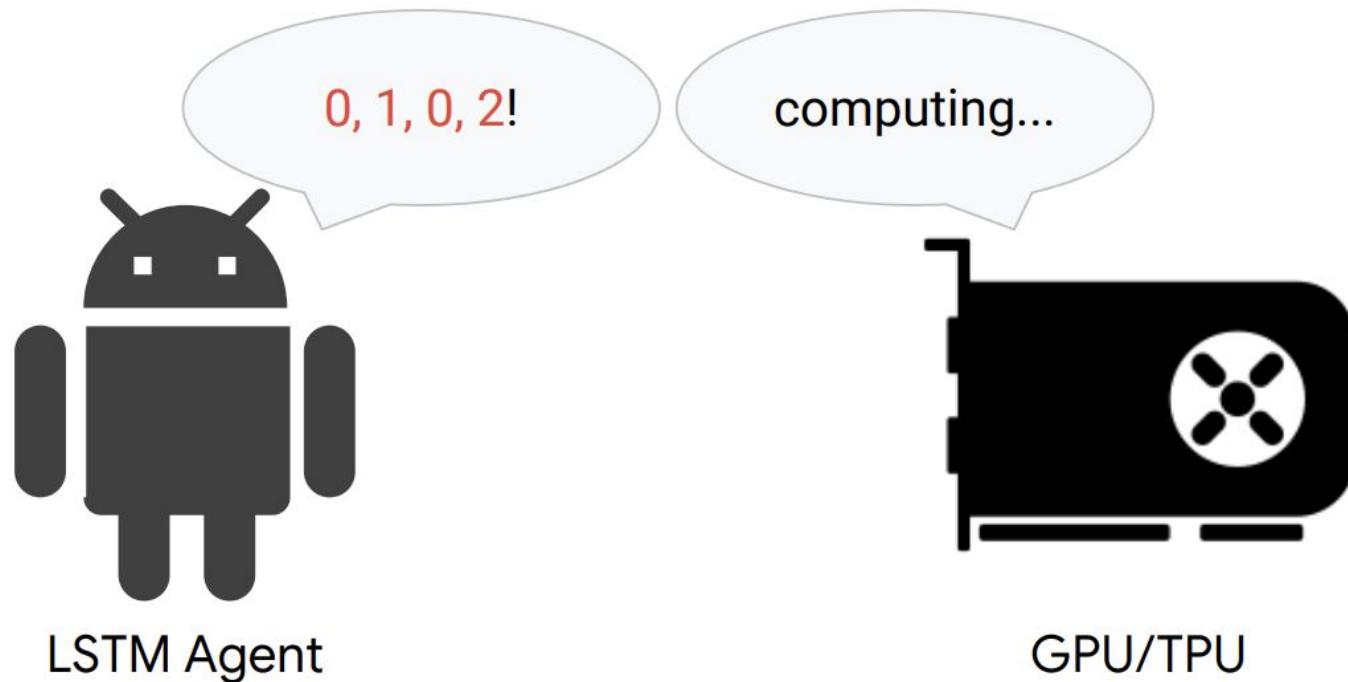
# AUTOMATIC ARCHITECTURAL SEARCH: WHAT TO SEARCH?

- Hyper-parameters e.g. related to learning rate, hidden size
- But for a **neural network**, many more lie in its **architecture**

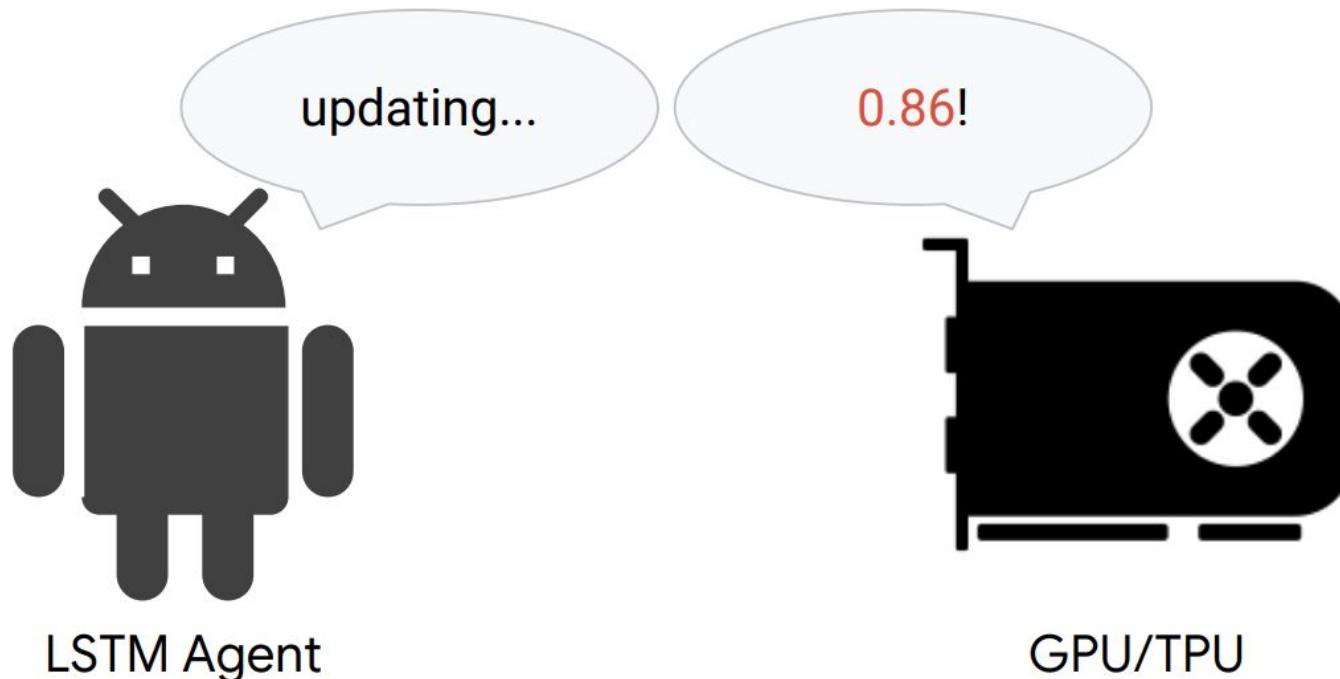


Can we **design network architectures automatically**, instead of relying on expert experience and knowledge?

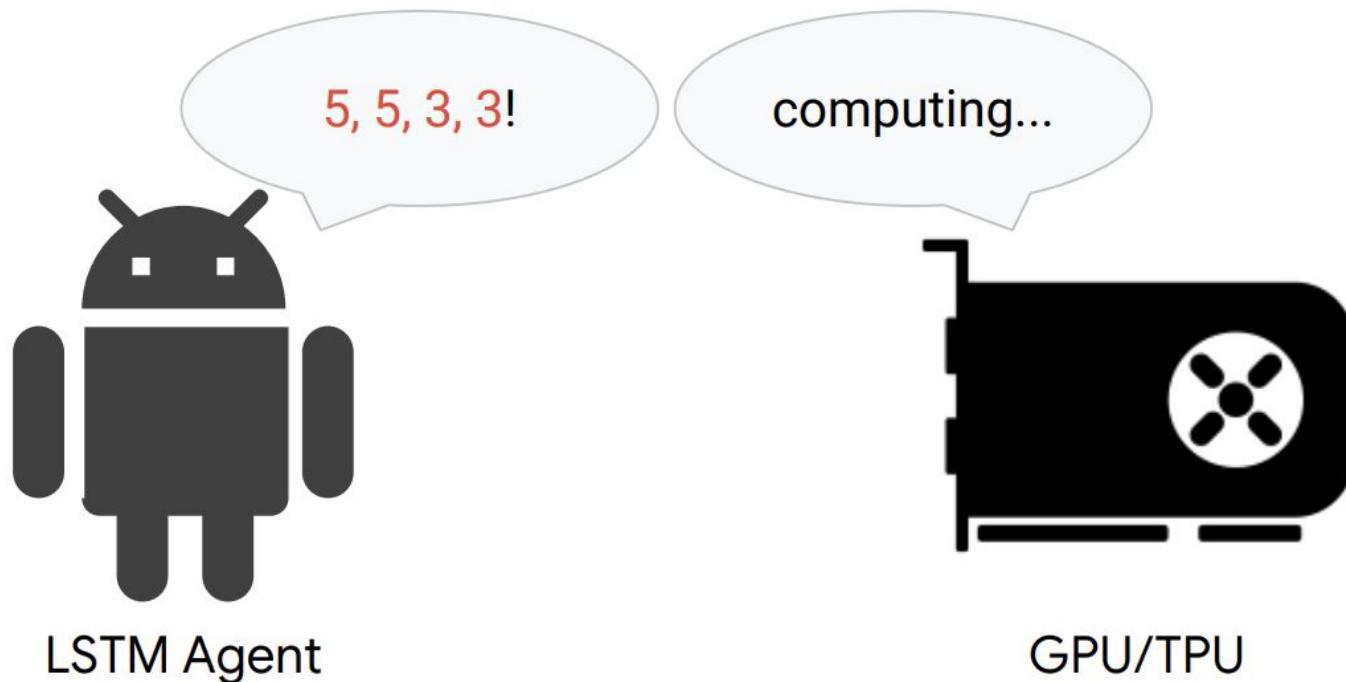
# REINFORCEMENT LEARNING: TRIAL AND ERROR



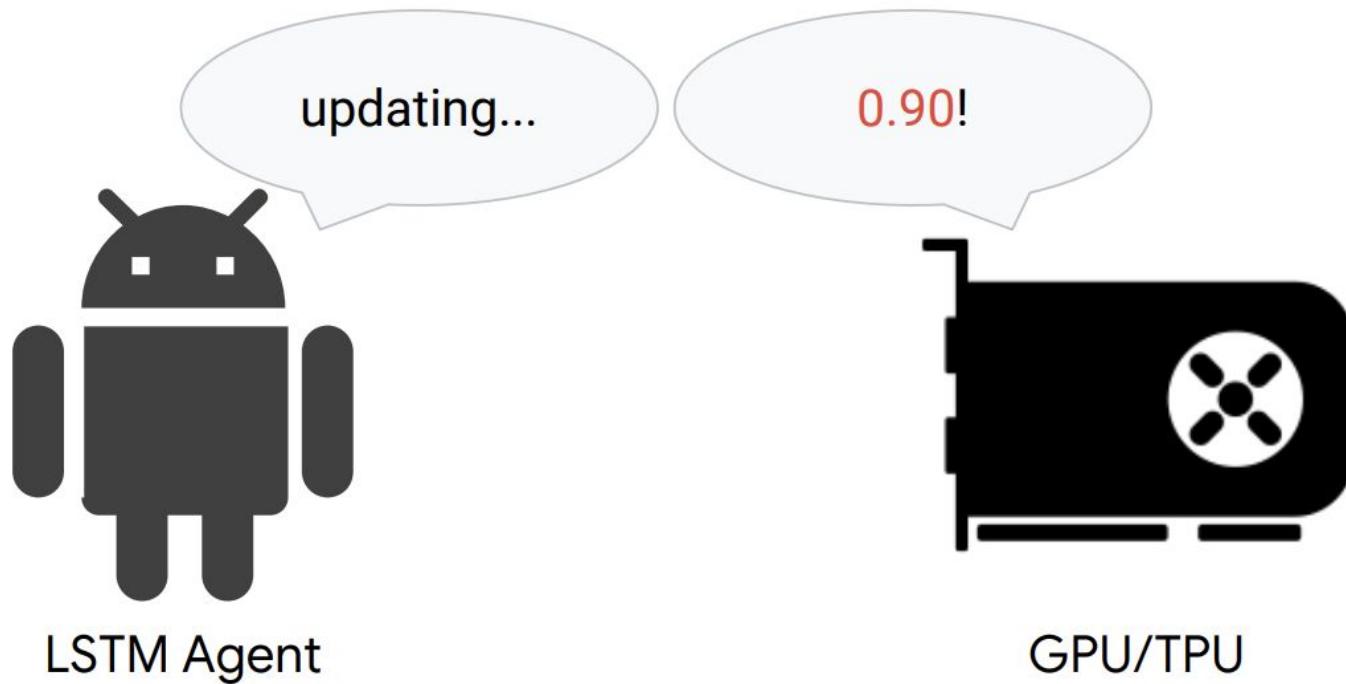
# REINFORCEMENT LEARNING: TRIAL AND ERROR



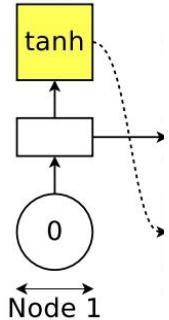
# REINFORCEMENT LEARNING: TRIAL AND ERROR



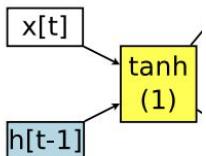
# REINFORCEMENT LEARNING: TRIAL AND ERROR



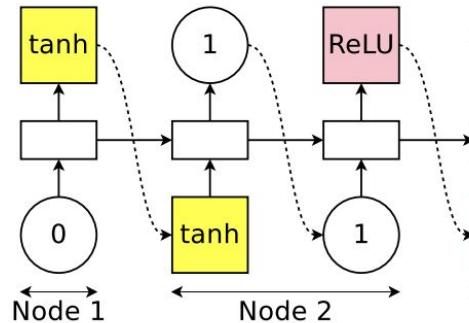
# ENAS-LSTM AGENT CREATING RNN CELL



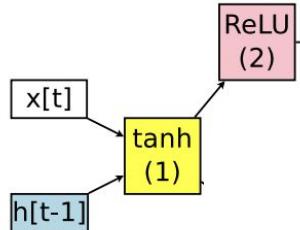
1. At node 1: The controller first samples an activation function. In our example, the controller chooses the tanh activation function, which means that node 1 of the recurrent cell should compute  $h_1 = \tanh(\mathbf{x}_t \cdot \mathbf{W}^{(x)} + \mathbf{h}_{t-1} \cdot \mathbf{W}_1^{(h)})$ .



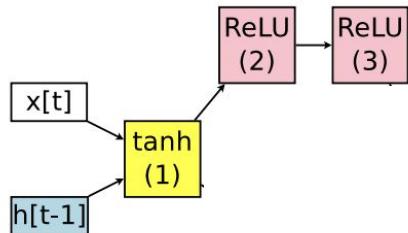
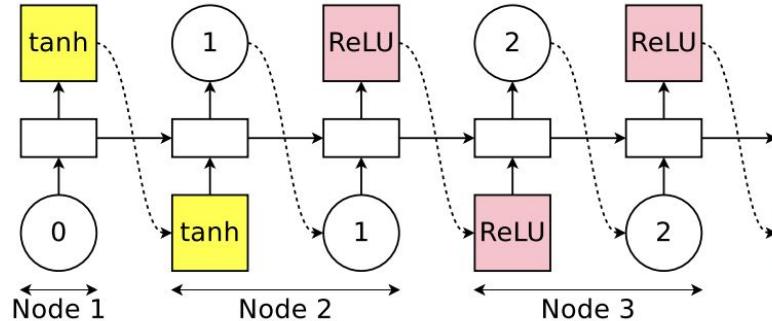
# ENAS-LSTM AGENT CREATING RNN CELL



1. At node 1: The controller first samples an activation function. In our example, the controller chooses the tanh activation function, which means that node 1 of the recurrent cell should compute  $h_1 = \tanh(\mathbf{x}_t \cdot \mathbf{W}^{(x)} + \mathbf{h}_{t-1} \cdot \mathbf{W}_1^{(h)})$ .
2. At node 2: The controller then samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function ReLU. Thus, node 2 of the cell computes  $h_2 = \text{ReLU}(h_1 \cdot \mathbf{W}_{2,1}^{(h)})$ .

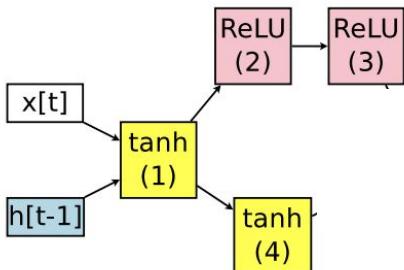
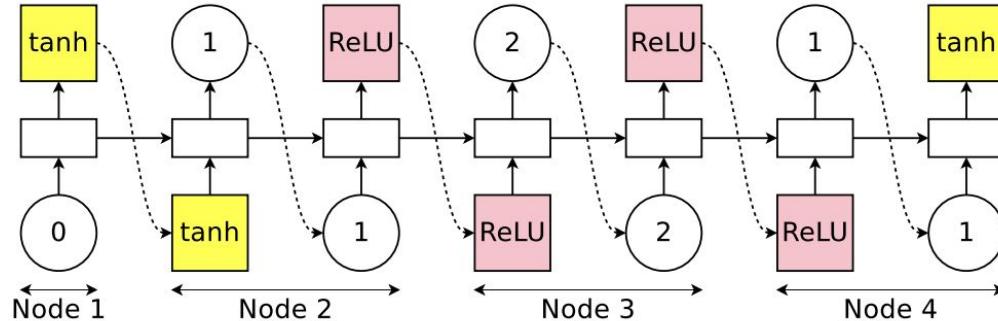


# ENAS-LSTM AGENT CREATING RNN CELL



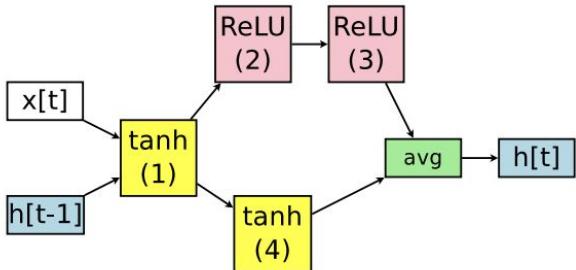
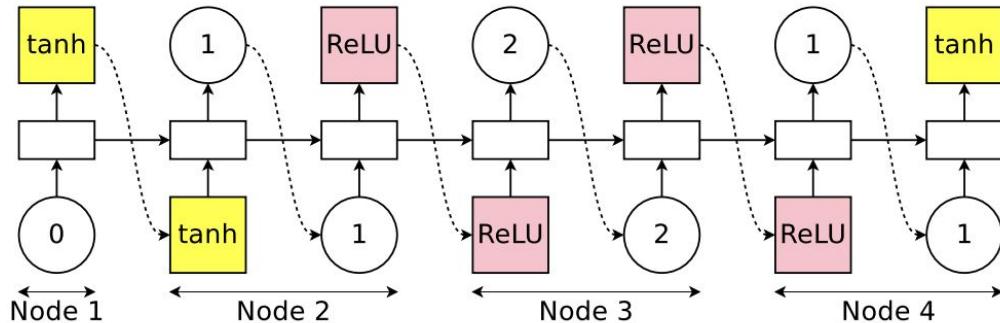
1. At node 1: The controller first samples an activation function. In our example, the controller chooses the tanh activation function, which means that node 1 of the recurrent cell should compute  $h_1 = \tanh(\mathbf{x}_t \cdot \mathbf{W}^{(x)} + \mathbf{h}_{t-1} \cdot \mathbf{W}_1^{(h)})$ .
2. At node 2: The controller then samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function ReLU. Thus, node 2 of the cell computes  $h_2 = \text{ReLU}(h_1 \cdot \mathbf{W}_{2,1}^{(h)})$ .
3. At node 3: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 2 and the activation function ReLU. Therefore,  $h_3 = \text{ReLU}(h_2 \cdot \mathbf{W}_{3,2}^{(h)})$ .

# ENAS-LSTM AGENT CREATING RNN CELL



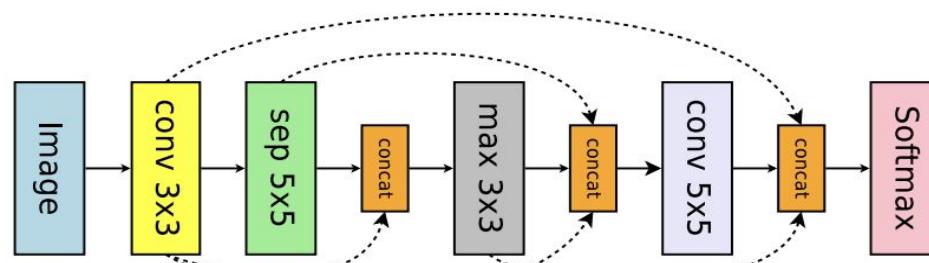
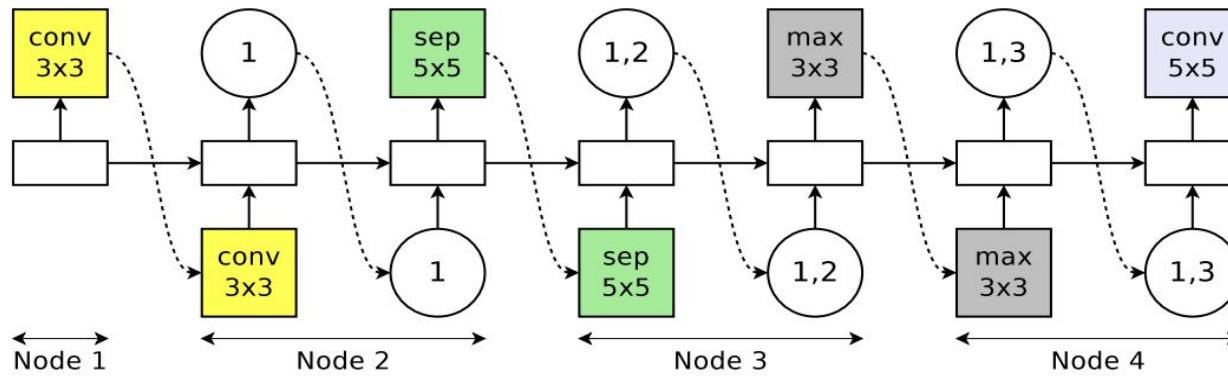
- At node 1: The controller first samples an activation function. In our example, the controller chooses the `tanh` activation function, which means that node 1 of the recurrent cell should compute  $h_1 = \tanh(\mathbf{x}_t \cdot \mathbf{W}^{(x)} + \mathbf{h}_{t-1} \cdot \mathbf{W}_1^{(h)})$ .
- At node 2: The controller then samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function `ReLU`. Thus, node 2 of the cell computes  $h_2 = \text{ReLU}(h_1 \cdot \mathbf{W}_{2,1}^{(h)})$ .
- At node 3: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 2 and the activation function `ReLU`. Therefore,  $h_3 = \text{ReLU}(h_2 \cdot \mathbf{W}_{3,2}^{(h)})$ .
- At node 4: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function `tanh`, leading to  $h_4 = \tanh(h_3 \cdot \mathbf{W}_{4,1}^{(h)})$ .

# ENAS-LSTM AGENT CREATING RNN CELL

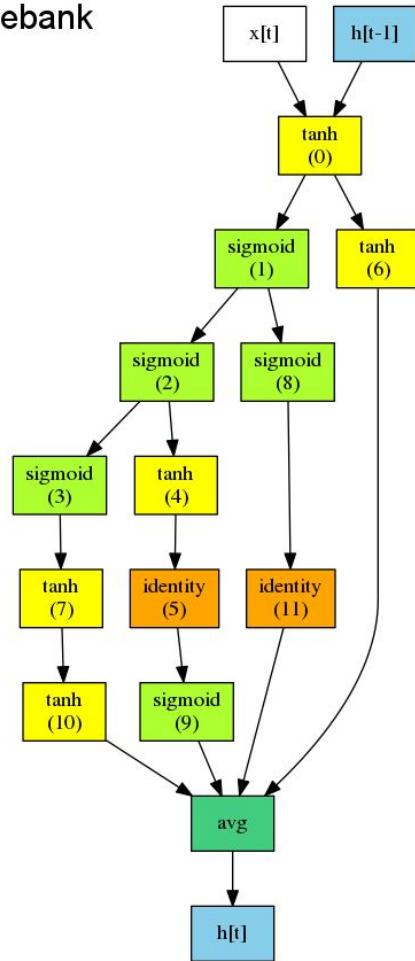


- At node 1: The controller first samples an activation function. In our example, the controller chooses the  $\text{tanh}$  activation function, which means that node 1 of the recurrent cell should compute  $h_1 = \tanh(\mathbf{x}_t \cdot \mathbf{W}^{(x)} + \mathbf{h}_{t-1} \cdot \mathbf{W}_1^{(h)})$ .
- At node 2: The controller then samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function  $\text{ReLU}$ . Thus, node 2 of the cell computes  $h_2 = \text{ReLU}(h_1 \cdot \mathbf{W}_{2,1}^{(h)})$ .
- At node 3: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 2 and the activation function  $\text{ReLU}$ . Therefore,  $h_3 = \text{ReLU}(h_2 \cdot \mathbf{W}_{3,2}^{(h)})$ .
- At node 4: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function  $\text{tanh}$ , leading to  $h_4 = \tanh(h_1 \cdot \mathbf{W}_{4,1}^{(h)})$ .
- For the output, we simply average all the loose ends, *i.e.* the nodes that are not selected as inputs to any other nodes. In our example, since the indices 3 and 4 were never sampled to be the input for any node, the recurrent cell uses their average  $(h_3 + h_4)/2$  as its output. In other words,  $\mathbf{h}_t = (h_3 + h_4)/2$ .

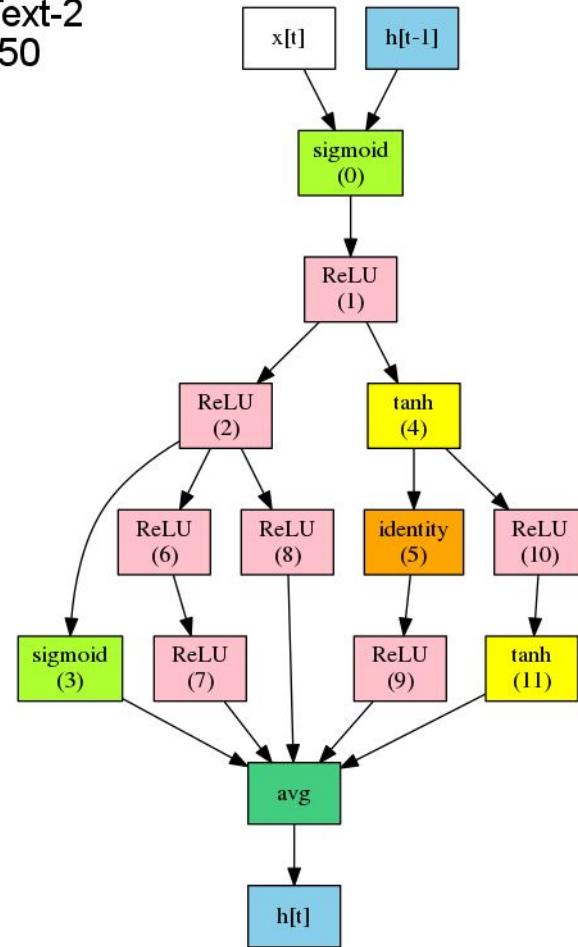
# SIMILARLY FOR CNNS



Penn Treebank  
step: 50



WikiText-2  
step: 50



# WHAT TO OPTIMIZE?

The LSTM controller act as a **policy network**.

So the optimization **alternate** the training of the architecture sampled from the policy (denoted by  $\mathbf{m}$ ) with parameter  $\omega$ . Using:

$$\nabla_{\omega} \mathbb{E}_{\mathbf{m} \sim \pi(\mathbf{m}; \theta)} [\mathcal{L}(\mathbf{m}; \omega)] \approx \frac{1}{M} \sum_{i=1}^M \nabla_{\omega} \mathcal{L} (\mathbf{m}_i, \omega)$$

And then policy optimization, using standard REINFORCE (William 1992) algorithm.

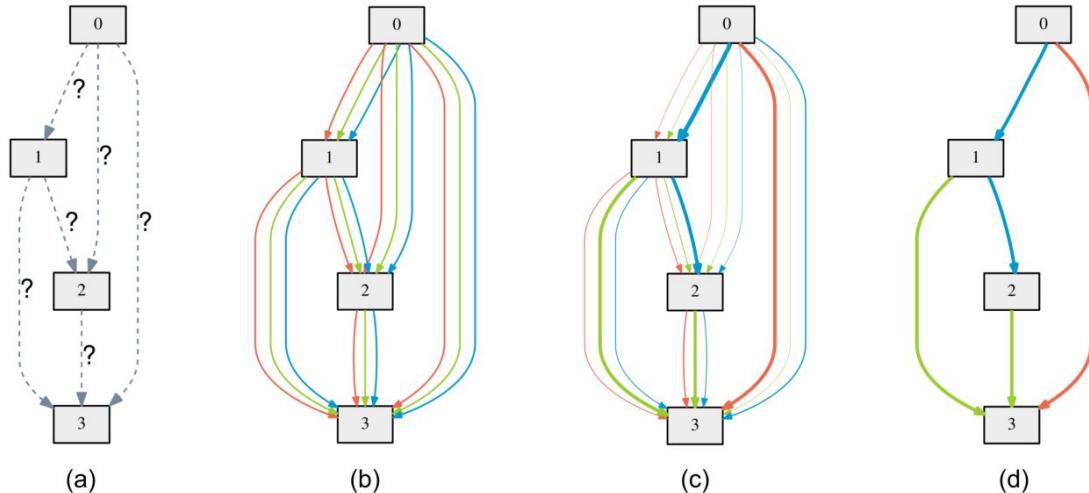
$$\mathbb{E}_{\mathbf{m} \sim \pi(\mathbf{m}; \theta)} [\mathcal{R}(\mathbf{m}, \omega)]$$

DARTS

# DIFFERENTIABLE ARCHITECTURE SEARCH METHOD (DARTS)

A gradient-based optimization method to efficiently search the best model architecture by relaxing the discrete search space to continuous space.

- No controllers
- No hyper-parameters
- No predictors
- Differentiable



# SEARCH SPACE

Lets define a computation cell as the building block as an acyclic graph.

$$x^{(i)} = \sum_{j < i} o^{(i,j)}(x^{(j)})$$

Where  $x^{(i)}$  is the node (e.g. a feature map in convolutional networks) and  $o^{(i,j)}$  is the operation (edge)

# WITHIN A CELL

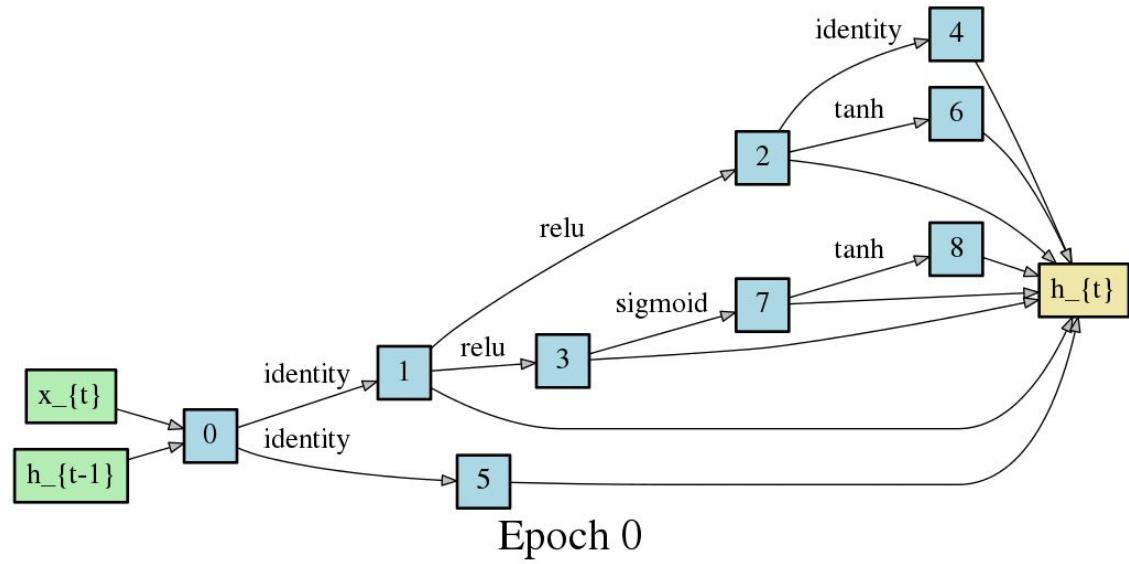
## Operations

### CNN:

- 'max\_pool\_3x3',
- 'avg\_pool\_3x3',
- 'skip\_connect',
- 'sep\_conv\_3x3',
- 'sep\_conv\_5x5',
- 'dil\_conv\_3x3',
- 'Dil\_conv\_5x5'

### RNN:

- 'none',
- 'tanh',
- 'relu',
- 'sigmoid',
- 'identity'



# CONTINUOUS RELAXATION AND OPTIMIZATION

Let  $\mathcal{O}$  be a set of candidate operations. Softmax function is applied over all possible operation.

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^{(i,j)})} o(x)$$

where each operation represents some function  $o(\cdot)$  to be applied to  $x^{(i)}$ .  
The operation mixing weights for a pair of nodes  $(i, j)$  are parameterized by a vector  $\alpha^{(i,j)}$  of dimension  $|\mathcal{O}|$

# CONTINUOUS RELAXATION AND OPTIMIZATION

At the end of search, a discrete architecture is obtained by replacing each mixed operation

$$o^{(i,j)} = \operatorname{argmax}_{o \in \mathcal{O}} \alpha_o^{(i,j)}$$

Analogous to architecture search using RL or evolution where the validation set performance is treated as the reward or fitness, **DARTS aims to optimize the validation loss, but using gradient descent.**

# BILEVEL OPTIMIZATION

Denote  $\mathcal{L}_{train}$  and  $\mathcal{L}_{val}$  the training and the validation loss. The goal for architecture search is to find  $\alpha^*$ , minimizes the validation loss  $\mathcal{L}_{val}(w^*, \alpha^*)$

$$\begin{aligned} \min_{\alpha} \quad & \mathcal{L}_{val}(w^*(\alpha), \alpha) \\ \text{s.t.} \quad & w^*(\alpha) = \operatorname{argmin}_w \mathcal{L}_{train}(w, \alpha) \end{aligned}$$

keeping the weights  $w_k$  fixed, we update the architecture so as to minimize the validation loss after a single step of gradient descent w.r.t. the weights:

$$\mathcal{L}_{val}(w_k - \xi \nabla_w \mathcal{L}_{train}(w_k, \alpha_{k-1}), \alpha_{k-1})$$

# APPROXIMATION

The gradient contains a matrix-vector product in its second term, which is expensive to compute.

$$\mathcal{L}_{val}(w_k - \xi \nabla_w \mathcal{L}_{train}(w_k, \alpha_{k-1}), \alpha_{k-1})$$

The gradient can be substantially reduced from  $O(|\alpha| |w|)$  to  $O(|\alpha| + |w|)$  using the finite difference approximation using Taylor Series.

$$\nabla_{\alpha, w}^2 \mathcal{L}_{train}(w, \alpha) \nabla_{w'} \mathcal{L}_{val}(w', \alpha) \approx \frac{\nabla_{\alpha} \mathcal{L}_{train}(w^+, \alpha) - \nabla_{\alpha} \mathcal{L}_{train}(w^-, \alpha)}{2\epsilon}$$

$$w^+ = w + \epsilon \nabla_{w'} \mathcal{L}_{val}(w', \alpha) \quad w^- = w - \epsilon \nabla_{w'} \mathcal{L}_{val}(w', \bar{\alpha}).$$

---

**Algorithm 1:** DARTS – Differentiable Architecture Search

---

Create a mixed operation  $\bar{o}^{(i,j)}$  parametrized by  $\alpha^{(i,j)}$  for each edge  $(i, j)$

**while** *not converged* **do**

- 1. Update weights  $w$  by descending  $\nabla_w \mathcal{L}_{train}(w, \alpha)$
- 2. Update architecture  $\alpha$  by descending  $\nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$

Replace  $\bar{o}^{(i,j)}$  with  $o^{(i,j)} = \text{argmax}_{o \in \mathcal{O}} \alpha_o^{(i,j)}$  for each edge  $(i, j)$

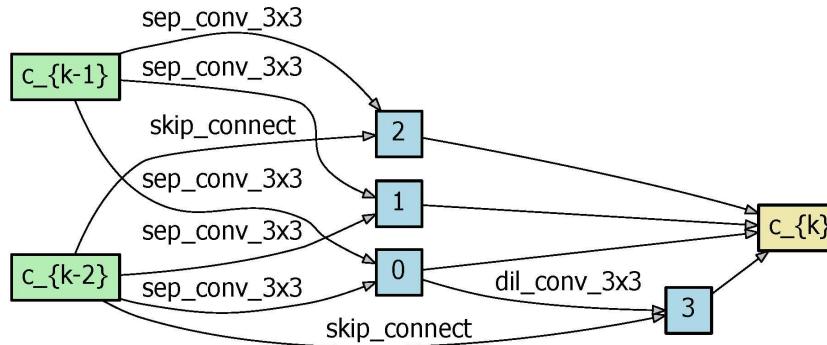
---

# ARCHITECTURE

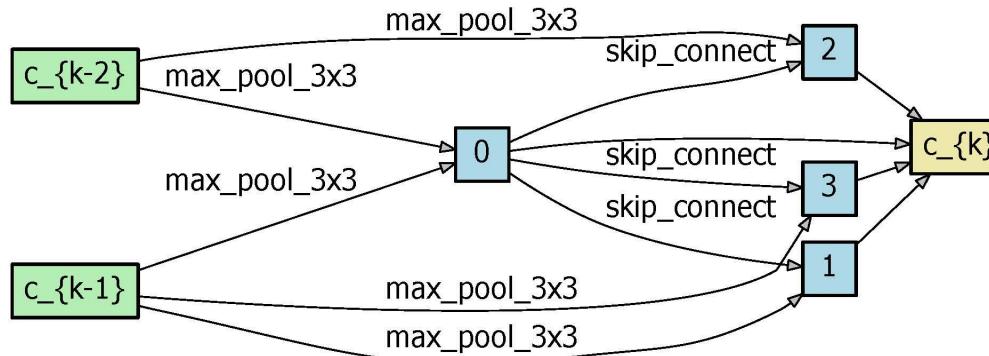
The first and second nodes of **cell k** are set to the outputs of **cell k-2** and **cell k-1**. Cells located at the 1/3 and 2/3 of the total depth of the network are reduction cells.

$(\alpha_{normal}, \alpha_{reduction})$  are shared in normal and reduction cells.

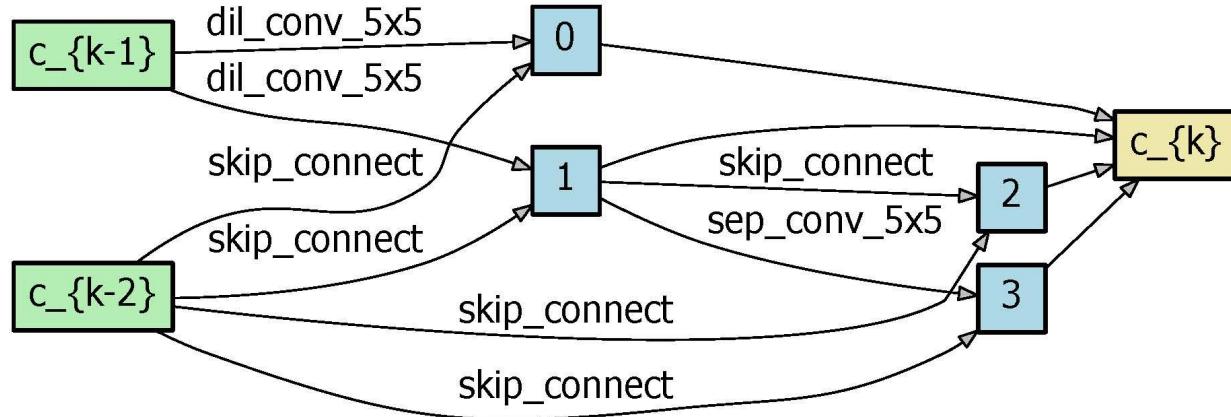
## Normal cell on ImageNet



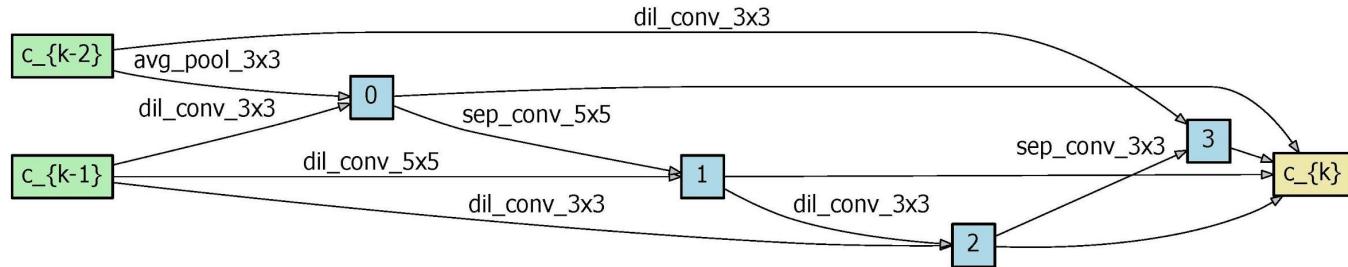
## Reduction cell on ImageNet



## Normal cell on Nexperia



## Reduction cell on Nexperia



CAM-GRADCAM

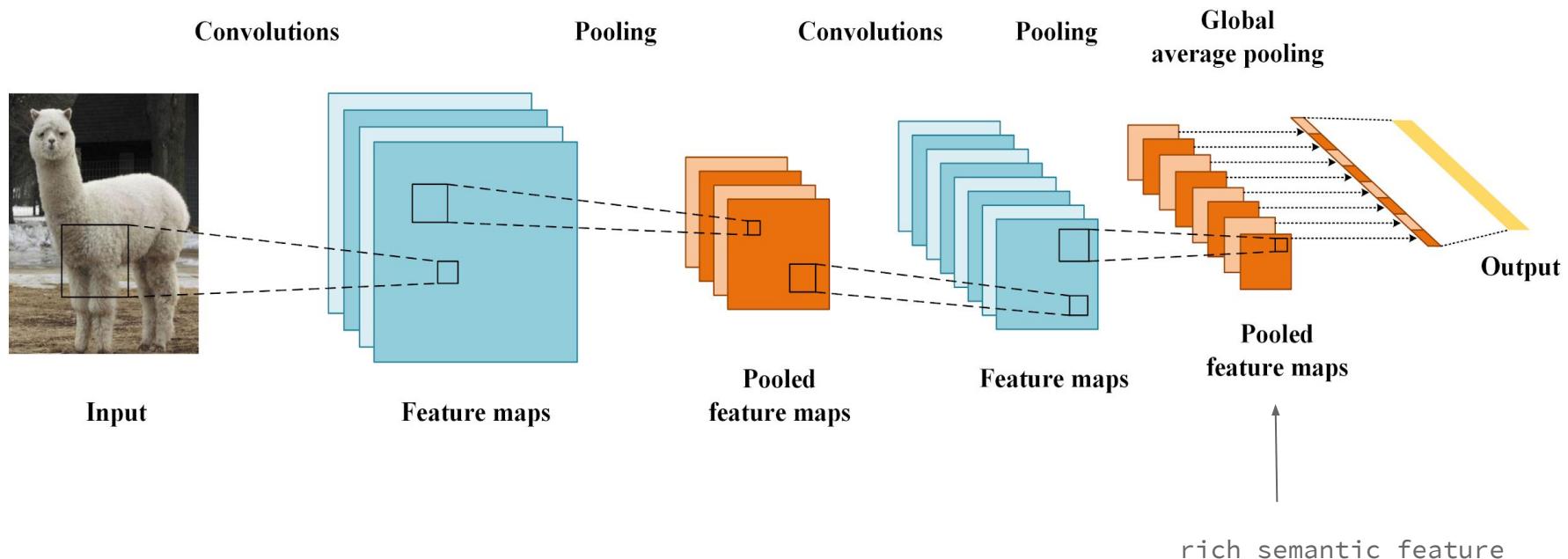
# VISUALIZATION (CAM VS GRAD-CAM)

How to interpret the result we get?

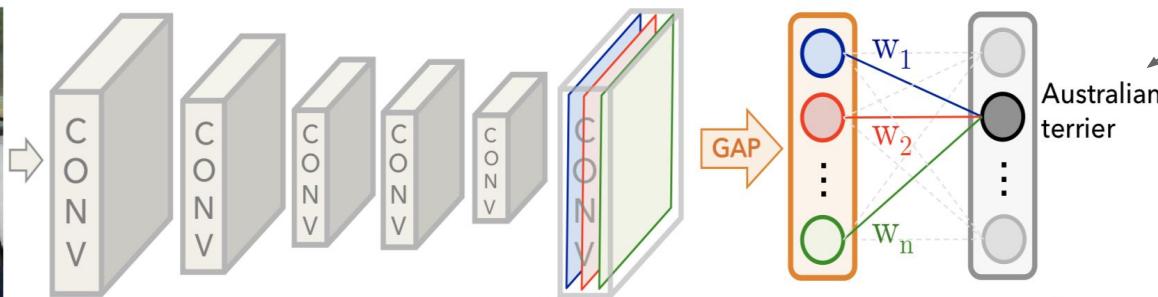
How to make sure CNN learn to extract correct features?

How to visualize the “attention” of CNN?

# VISUALIZATION (CAM VS GRAD-CAM)

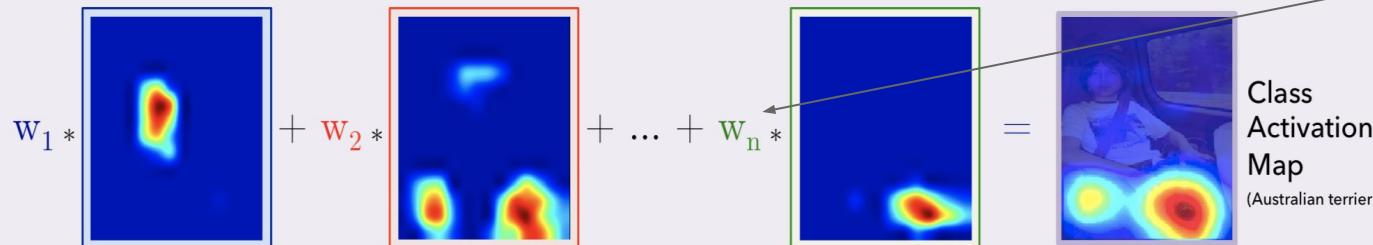


# CLASS ACTIVATION CAPPING - CAM



Why  
Australian  
terrier?

Class Activation Mapping



$w$  tell you  
which feature  
is more  
important

# CLASS ACTIVATION CAPPING - CAM

$$y^c = \sum_k w_k^c \underbrace{\frac{1}{Z} \sum_i \sum_j}_\text{global average pooling} A_{ij}^k$$

class feature weights                              feature map

**Shortcoming:** need to be re-trained, hard to visualize when retraining is very expensive

$$L_{\text{CAM}}^c = \underbrace{\sum_k w_k^c A^k}_\text{linear combination} .$$

# GRADIENT-WEIGHTED CLASS ACTIVATION CAPPING (GRAD-CAM)

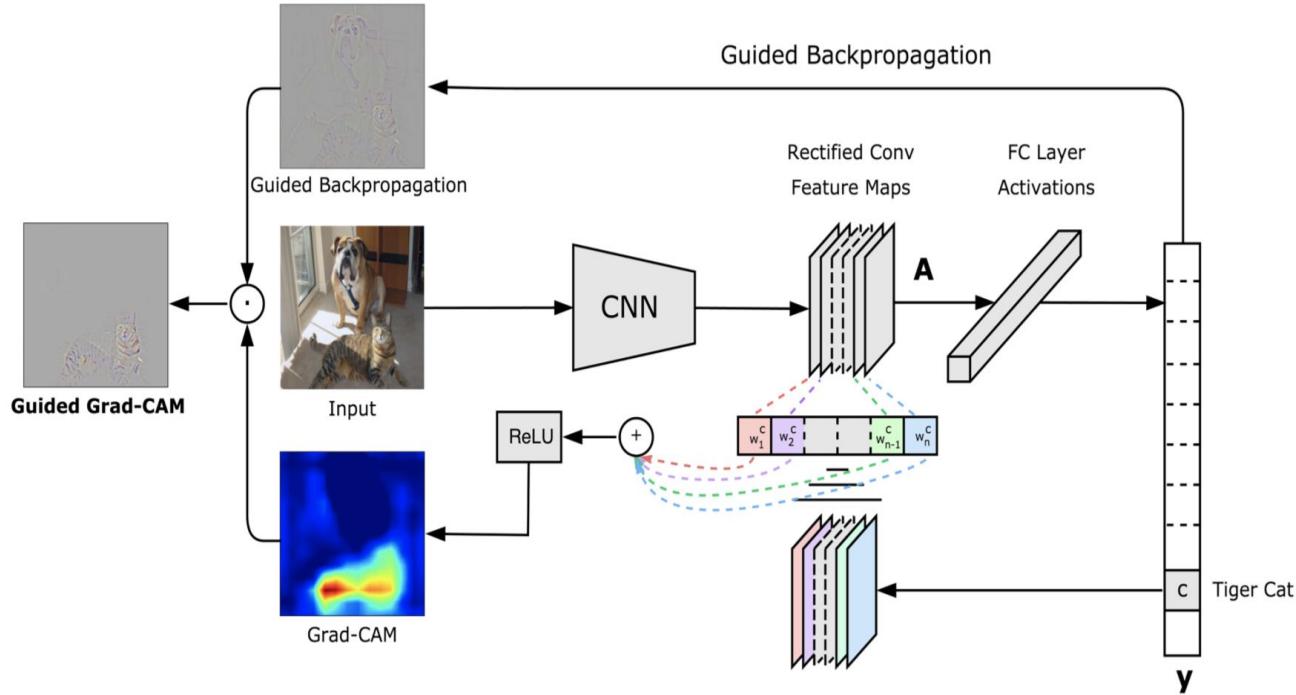
$$\alpha_k^c = \underbrace{\frac{1}{Z} \sum_i \sum_j}_{\text{global average pooling}} \underbrace{\frac{\partial y^c}{\partial A_{ij}^k}}_{\text{gradients via backprop}}$$

$$L_{\text{Grad-CAM}}^c = \text{ReLU} \left( \underbrace{\sum_k \alpha_k^c A^k}_{\text{linear combination}} \right)$$

Instead of obtain feature map weights from retrained linear layer, Grad-CAM **compute** the **gradient** of the  $y$  with respect to feature maps  $A$ .

Use **ReLU** to **normalize** the weighted combination of feature maps. The motivation for the **ReLU** is the following: we are only interested in the features that have a **positive influence** on the class of interest

# GRADIENT-WEIGHTED CLASS ACTIVATION CAPPING (GRAD-CAM)



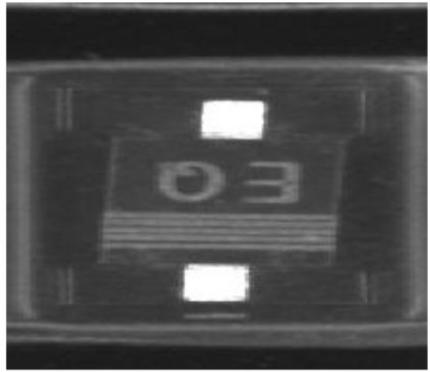


Figure 6: Sample labeled as good in the training set

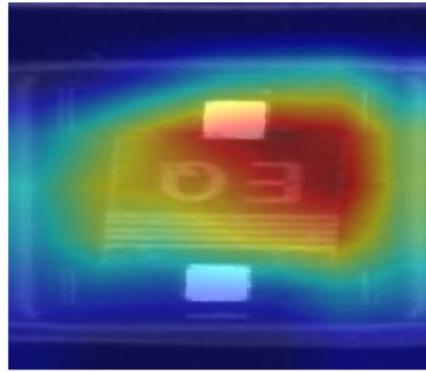


Figure 7: After backpropagation using correct labels

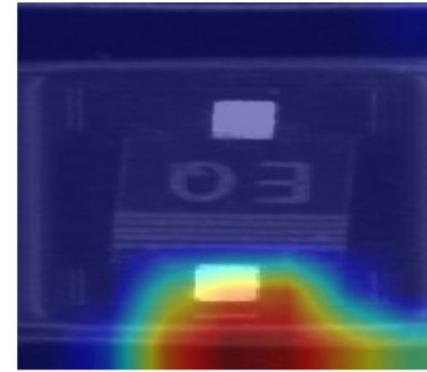


Figure 8: After backpropagation using wrong labels

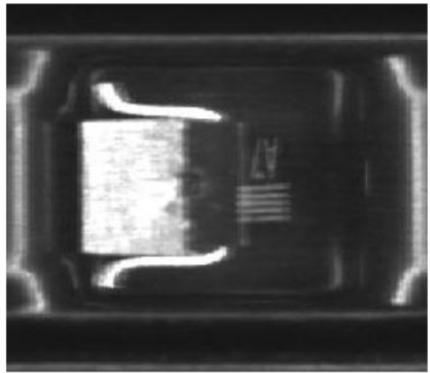


Figure 9: Sample labeled as bad in the training set

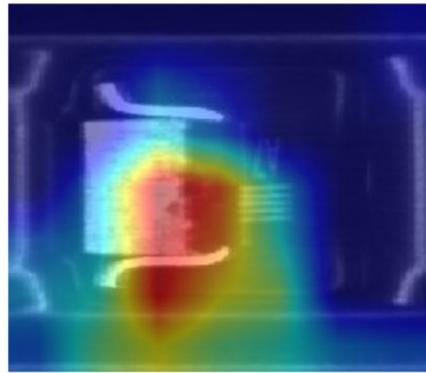


Figure 10: After backpropagation using correct labels

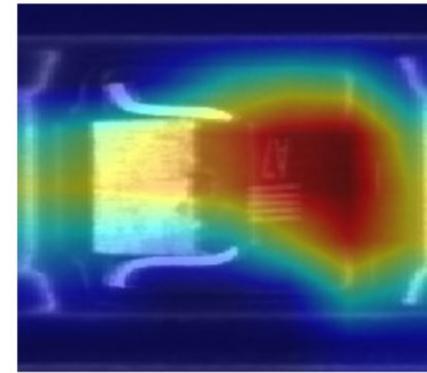


Figure 11: After backpropagation using wrong labels

THANKS

Q&A