Lecture 7:
Tricks of the trade

Advanced deep learning

Rémy Sun
remy.sun@inria.fr

Course organization

Advanced Deep Learning

UNIVERSITÉ CÔTE D'AZUR

- Goal: In-depth understanding of important Deep Learning staples
  - Reinforce what you have already seen
  - Introduce state of the art models

- This is a hands-on course in pytorch
  - Minimal math
    - Enough to understand
  - Quite a bit of coding
  - Get comfortable with the standard pipeline

Course organization: 10 Lectures

UNIVERSITÉ CÔTE D'AZUR

- L1-2: Overview of Deep Learning (F. Precioso)
- L3-4: Fundamentals of Deep Learning (R. Sun)
- L5-6: Transformers (R. Sun)

- *L8: Large models (LLMs, VLMs, Generators) (R. Sun)*

- **L7: Tricks of the trade (R. Sun)**

- L9: Ethics of AI (F. Precioso)

- L10: Intro to generative models (P-A. Mattei)

Today!

UNIVERSITÉ CÔTE D'AZUR

- Goal: Acquire a toolbox of things to try with networks
  - Understand what issues models might have
  - Know what tools to look for

- Companion lab session
  - Code a basic training codebase
    - With train/val/test split
  - Try a few things
    - E.g. Optimizers, activation functions, …

Before we start

How does pytorch work?

UNIVERSITÉ CÔTE D'AZUR

- Create neural network
  - Use the torch.nn modules
    - Can use torch.nn.Modules
    - Or create a new class inheriting from torch.nn modules

- The training loop is

How does pytorch work?

UNIVERSITÉ CÔTE D'AZUR

- Create neural network
  - Use the torch.nn modules
    - Can use torch.nn.Modules
    - Or create a new class inheriting from torch.nn modules

- The training loop is

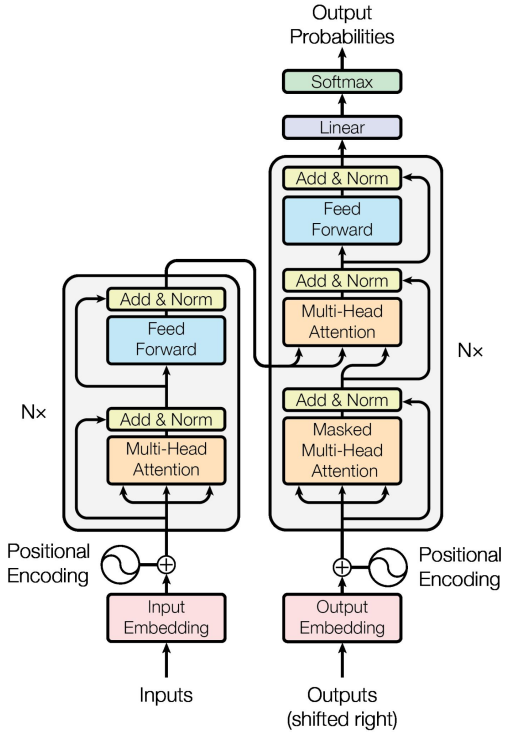
```
yhat = model(X)
L,acc = loss_accuracy(loss,yhat,Y)
optim.zero_grad()
L.backward()
optim.step()
```

Refresher on Transformers

A new type of neural layer for everying

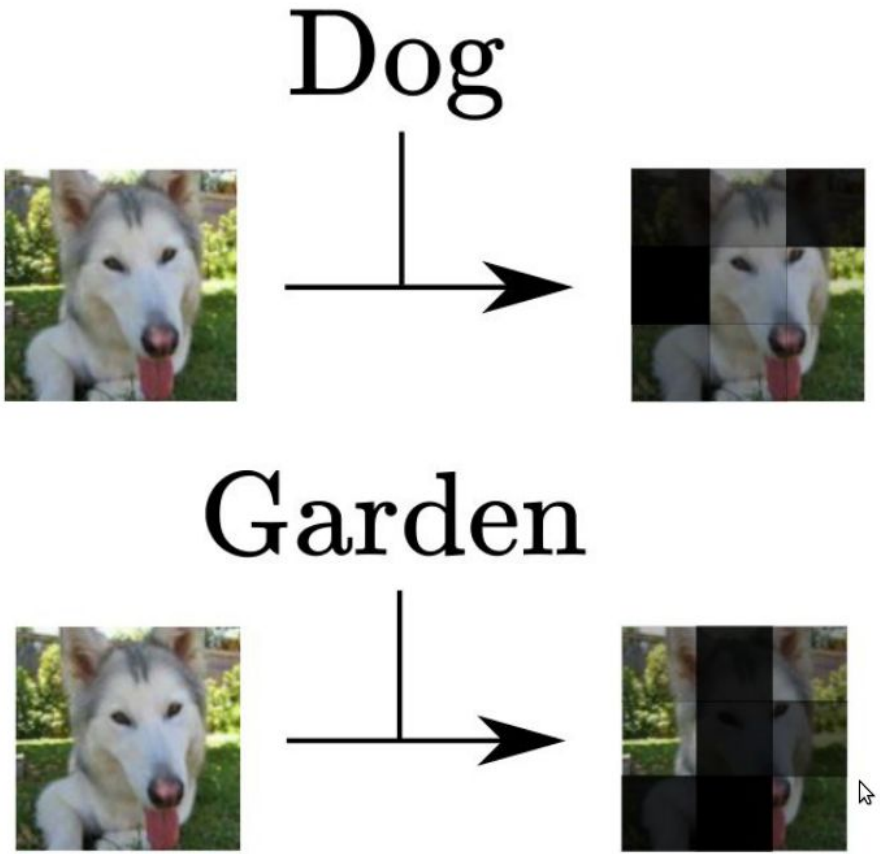
UNIVERSITÉ CÔTE D'AZUR

- What is the state of the art in
  - Computer vision?
    - ~~CNNs~~ -> **Transformers**
  - Natural language processing?
    - ~~RNNs~~ -> **Transformers**
  - Time series?
    - ~~RNNs or TCNs~~ -> **Transformers**
  - Multimodal problems?
    - ~~Hybrid?~~ -> **Transformers**



- Transformers use keeps increasing over time

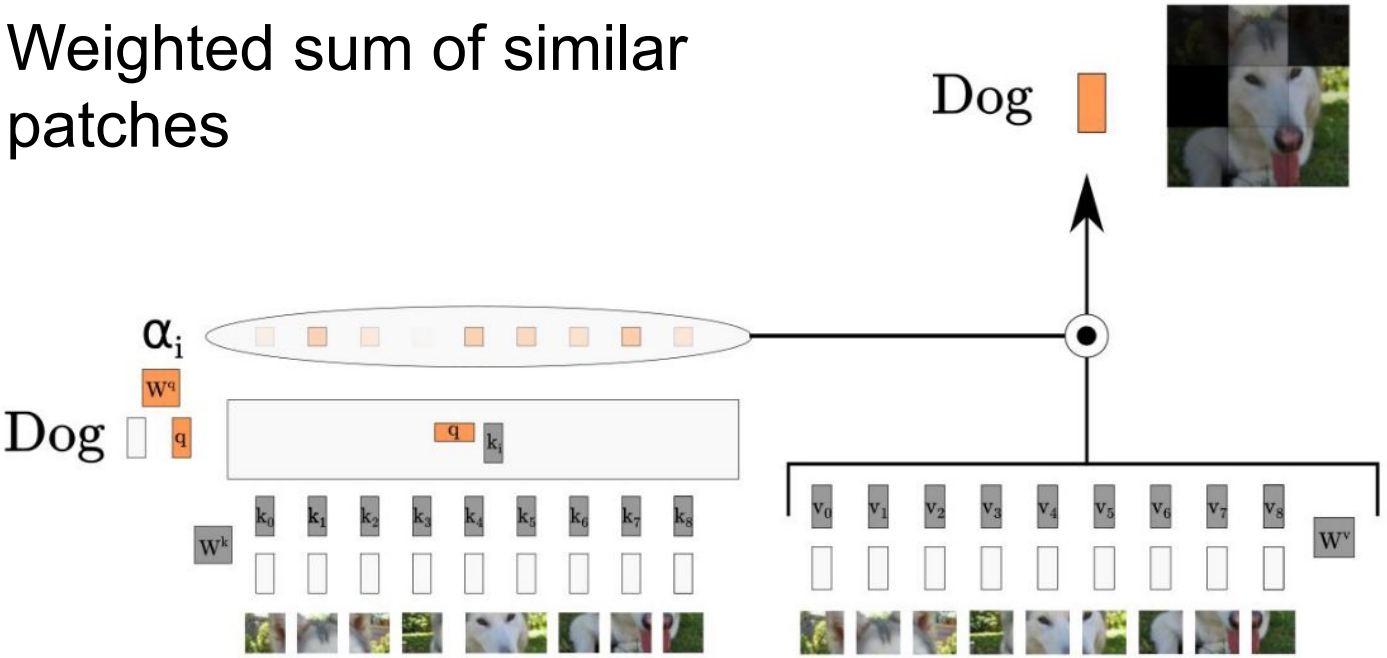
Main idea



- What is a Dog?
  - It is a 4 legged animal with fur and ears and eyes and a head and …
  - It is on this part of that picture.

Attended representation

- Weighted sum of similar patches



Attended representation

```python
def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]

    ########################
    ### YOUR CODE HERE! ###
    ########################

    # Compute attn_logits
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits /= math.sqrt(d_k)

    # Apply mask if not None
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, - 1e14)

    # Pass through softmax
    attention = F.softmax(attn_logits, dim=-1)

    # Weight values accordingly
    output_values = torch.matmul(attention, v)

    ########################
    ###       END        ###
    ########################

    return output_values, attention
```

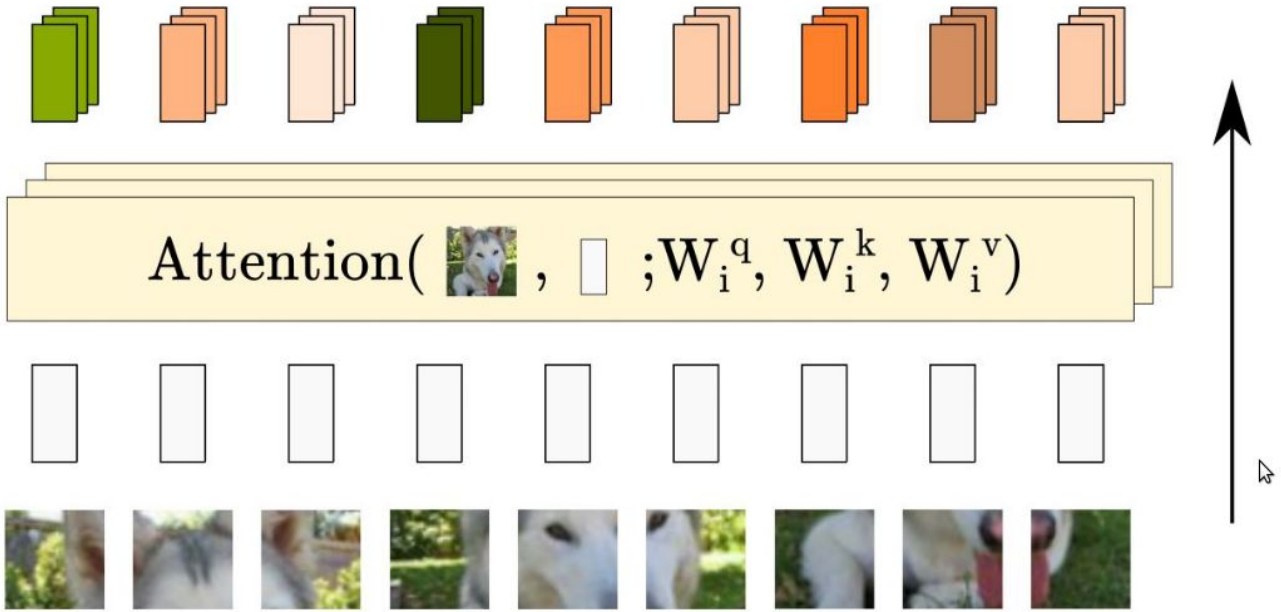
```python
def forward(self, x, mask=None, return_attention=False):

    ########################
    ### YOUR CODE HERE! ###
    ########################

    batch_dim, seq_length, input_dim = x.shape

    # Compute linear projection for qkv and separate heads
    # QKV: [Batch, Head, SeqLen, Dims]
    qkv = self.qkv_proj(x) # Batch x SeqLen x Hidden_dim * 3
    qkv = qkv.reshape(batch_dim, seq_length, self.num_heads, 3* self.head_dim)
    qkv = qkv.permute(0, 2, 1, 3)
    q, k, v = qkv.chunk(3, dim=-1)


    # Apply Dot Product Attention to qkv ()
    attention_values, attention = scaled_dot_product(q, k, v)

    # Concatenate heads to [Batch, SeqLen, Embed Dim]
    attention_values = attention_values.reshape(batch_dim, seq_length, self.embed_dim)

    # Output projection
    o = self.o_proj(attention_values)

    ########################
    ###       END        ###
    ########################

    if return_attention:
        return o, attention
    else:
        return o
```

Self-attention

- You can compare a sequence to itself!

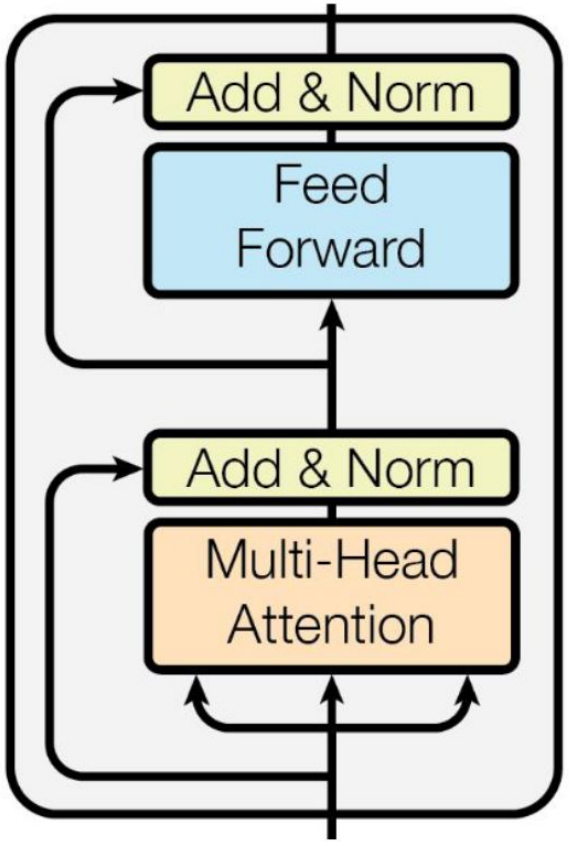
MultiHead

- And even have multiple interpretations



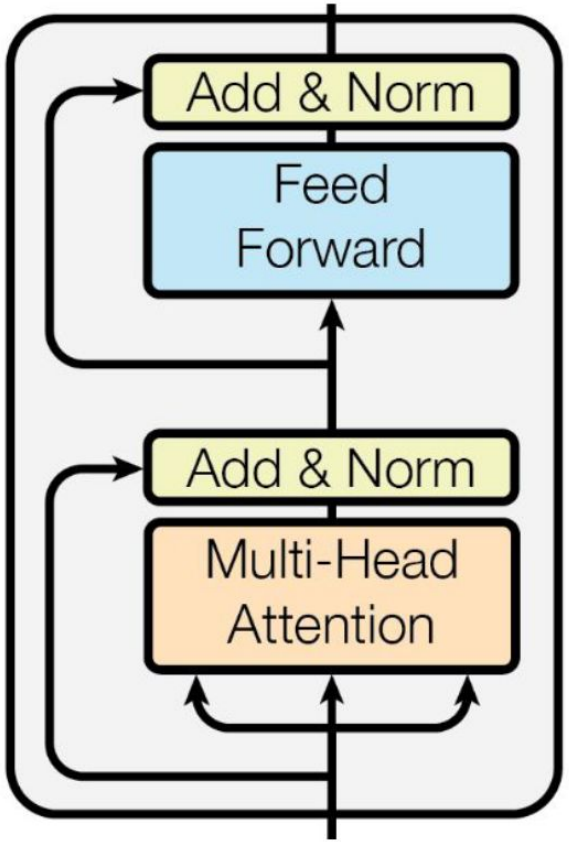
Transformer (Encoder) Block

- Basic encoder block
  - MultiHead attention
  - Skip connection
  - Small MLP applied to each token
  - Skip connection
- Stacked in a transformer



Transformer (Encoder) Block

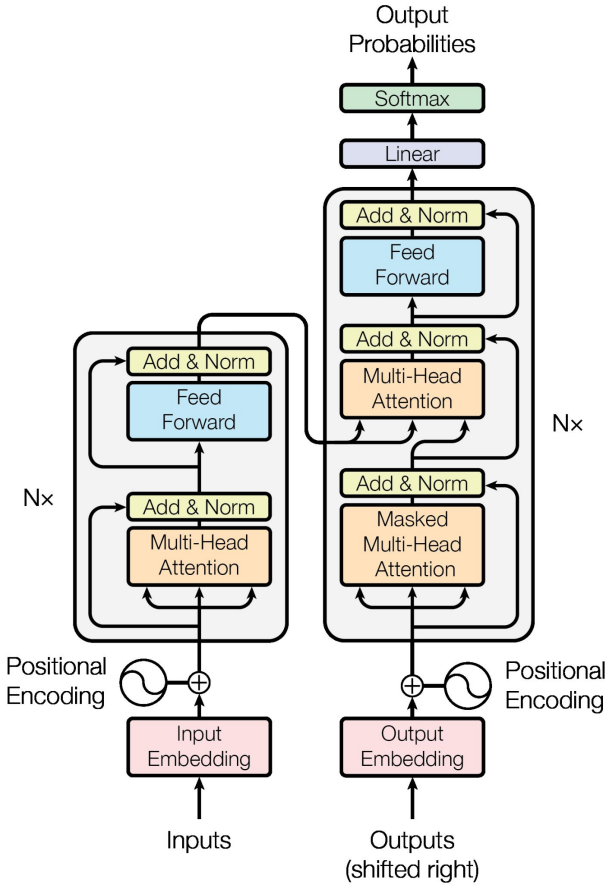
```python
class EncoderBlock(nn.Module):
    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Args:
            input_dim: Dimensionality of the input
            num_heads: Number of heads to use in the attention block
            dim_feedforward: Dimensionality of the hidden layer in the MLP
            dropout: Dropout probability to use in the dropout layers
        """
        super().__init__()

        # Create Attention layer
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

        # Create Two-layer MLP with droput
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, input_dim*2),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(2*input_dim, input_dim)
        )
        # Layers to apply in between the main layers (Layer Norm and Dropout)
        self.norm = nn.Sequential(
            nn.LayerNorm(input_dim),
            nn.Dropout(dropout)
        )

    def forward(self, x, mask=None):
        # Compute Attention part
        attn=self.self_attn(x)
        x=self.norm(attn+x)

        # Compute MLP part
        x = self.norm(x+self.mlp(x))

        return x
```



Seminal success: Vaswani et al. 2017

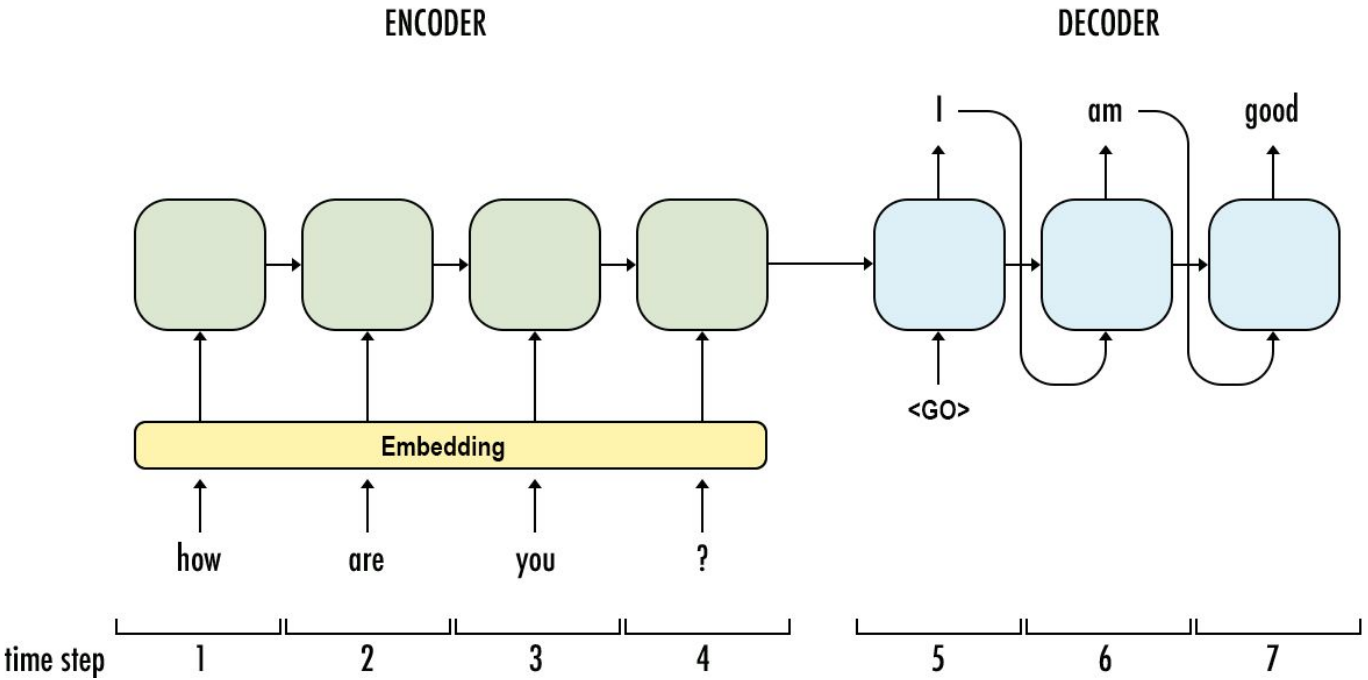
- Completely does away with recurrent units
  - Attention as a first class citizen!
  - Introduces element wise MLP for transform

- Transformer
  - Transforms the input throughout the layers
  - Also to blame for BERT, ELMO, DALL-E, …

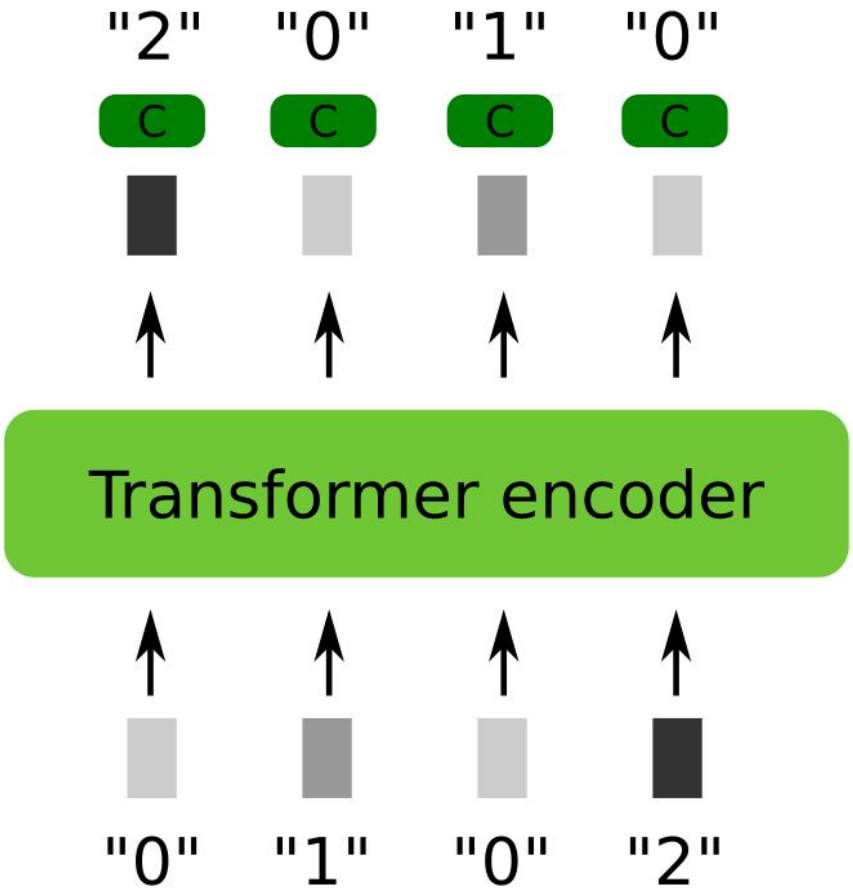


Autoregressive prediction

- Predict next token, feed it back into the decoder

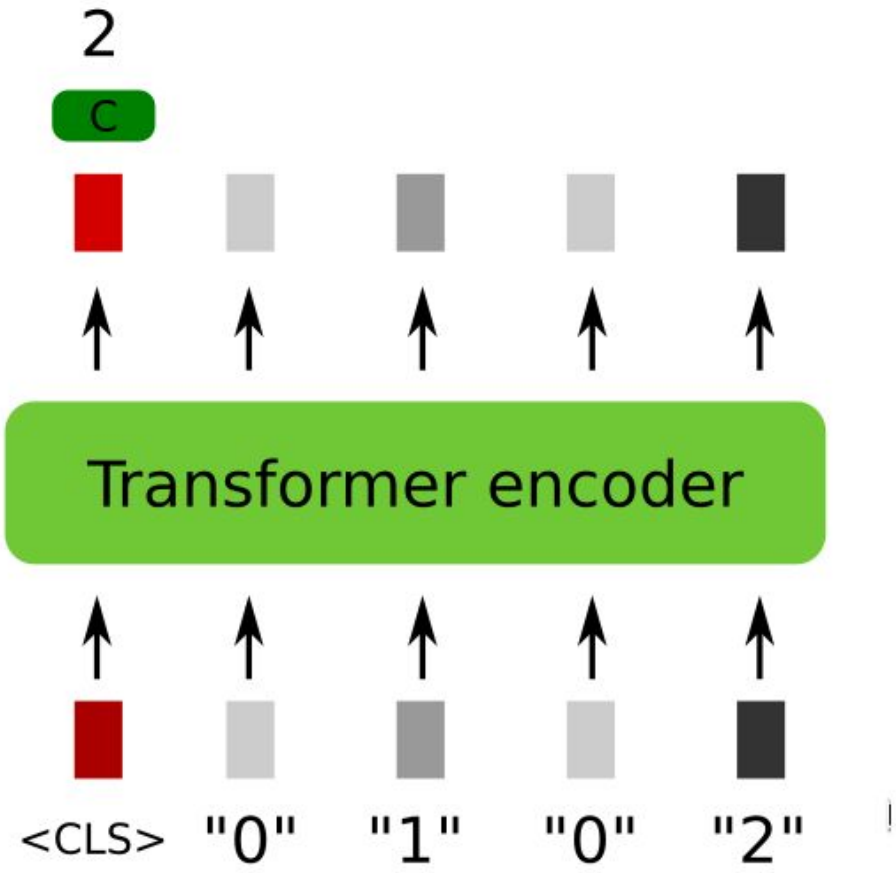


Token level predictions



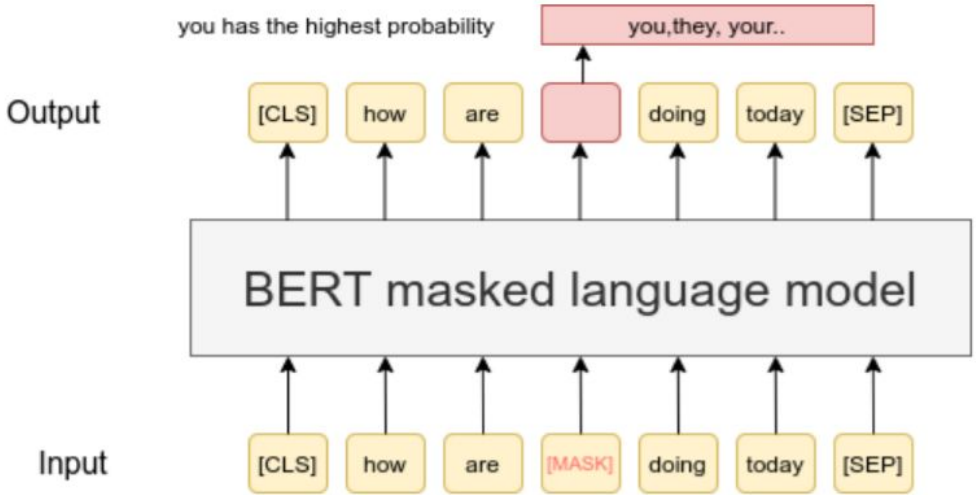
- Rich token features
  - Due to attention!
- Predictions for each token
  - Invert sequence here
- Predict with linear layers

Classification token



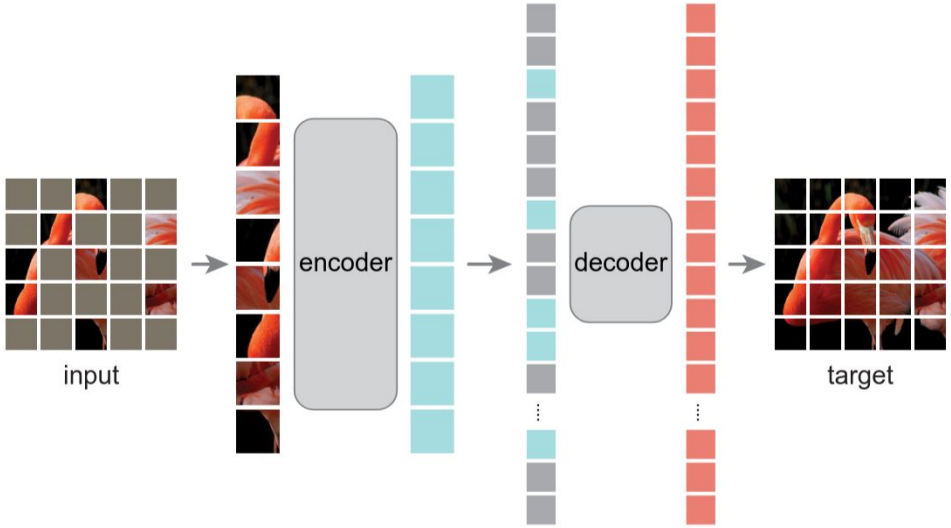
- Rich token features
  - Due to attention!
- Question/Classification token
  - “What is the class”
- Accumulate relevant info from other tokens

BERT pretraining



- Pseudo-objective
  - Mask part of the sentence
  - Try to predict the masked part

Masked auto-encoders



- Also works for images!
- Originally introduced in BeIT paper

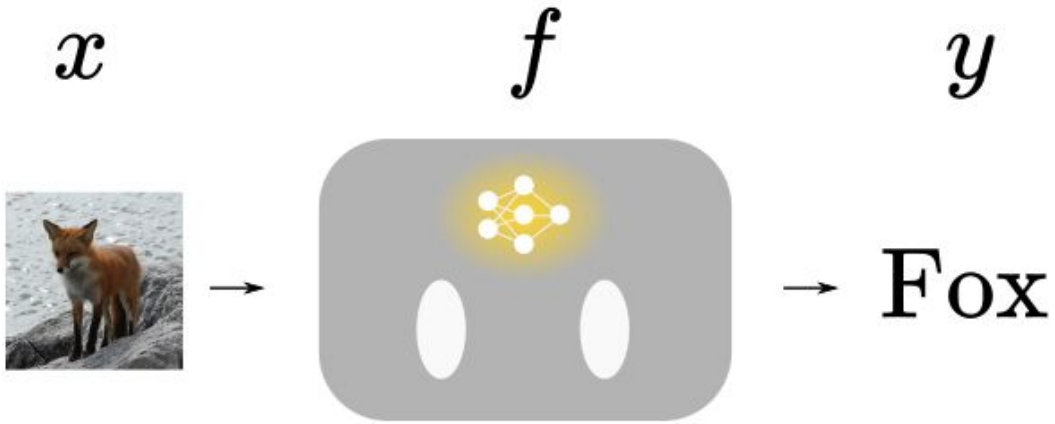
Vision Transformers as backbones

UNIVERSITÉ CÔTE D'AZUR

- Extract rich features from images
  - Including long range dependencies
- Masked token prediction for pretraining
  - Only works with transformers
  - Much better than traditional contrastive training
- Replace CNNs as backbones for downstream tasks
  - Segmentation
  - Detection

Let’s train a good model!

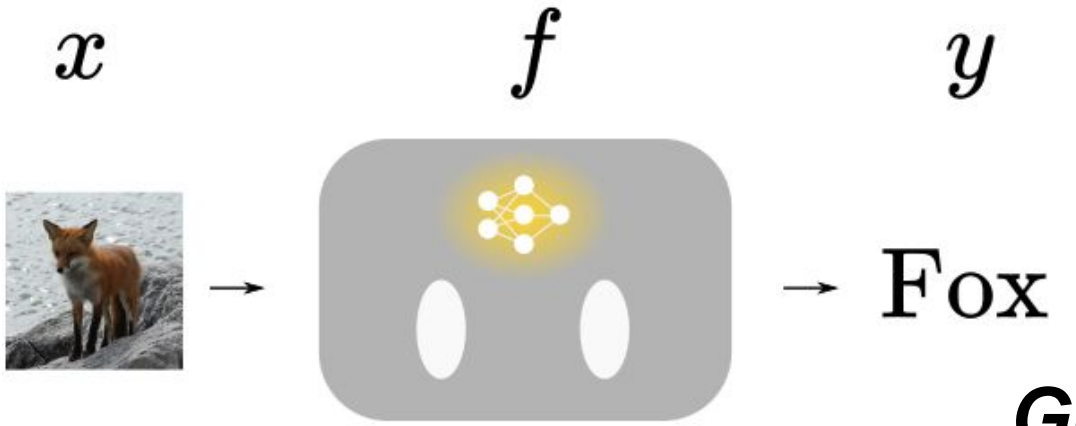
What is the goal with deep learning?



- Find (robot) f that classifies images well
  - Often based on neural networks

$$\forall (x, y) \in \mathcal{D}, f(x) = y$$

What is the goal with deep learning?



Get a good score!

- Find (robot) f that classifies images well
  - Often based on neural networks

$$\forall (x, y) \in \mathcal{D}, f(x) = y$$

What is a score?

- A score is a valuation of how good the model is
  - On some data we have

- A few classical scores
  - Accuracy (Classification)
  - Precision/Recall/F1 (Binary Retrieval)
  - Mean Squared Error (Regression)
  - Mean Average Precision (Retrieval/Detection)
  - Mean Intersection over Union (Segmentation)

Train / Test split

- What do we compute the score on?
  - Some Data

- What is “some data”?
  - Training dataset?

Train / Test split

- What do we compute the score on?
  - Some Data

- What is “some data”?
  - Training dataset?
    - **NO, being good on training means nothing!**
    - We want good performance in general
    - Need another dataset

Train / Test split

- What do we compute the score on?
  - Some Data

- What is “some data”?
  - Training dataset?
    - **NO, being good on training means nothing!**
    - We want good performance in general
    - Need another dataset
  - Testing dataset
    - Data not seen at training
    - Proxy for the problem distribution

Optimizing hyperparameters on the val set

UNIVERSITÉ CÔTE D'AZUR

- How do compare models and hyperparameters to find the best model to use?
  - We look at the score!
  - On the test set?

Optimizing hyperparameters on the val set

UNIVERSITÉ CÔTE D'AZUR

- How do compare models and hyperparameters to find the best model to use?
  - We look at the score!
  - On the test set?
    - **NO! Completely Illegal!**
    - Turns the test set into a pseudo train set

Optimizing hyperparameters on the val set

UNIVERSITÉ CÔTE D'AZUR

- How do compare models and hyperparameters to find the best model to use?
  - We look at the score!
  - On the test set?
    - **NO! Comple**
    - Turns the tes eudo train set

- Create a separate validation set
  - Classical train/val/test split
  - Can be split from the training set

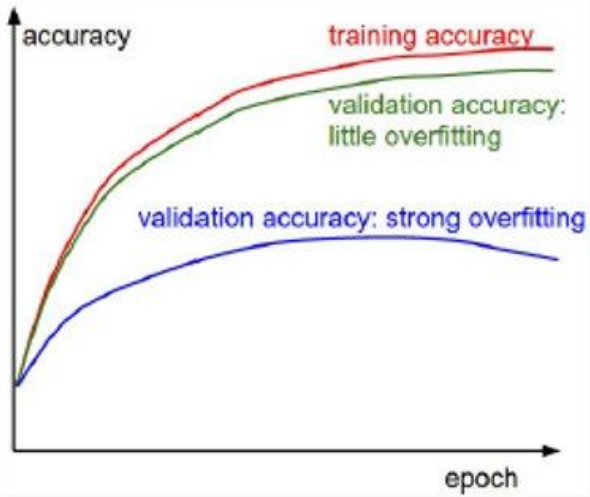
Split the train/val/test sets

UNIVERSITÉ CÔTE D'AZUR

- We only have a train/test set on the notebook!!!
  - Take Validation set from train set
  - Keep eyes of test set performance

- Pytorch Datasets/Dataloaders
  - Datasets give preprocessed samples/labels
  - Dataloaders are used to iteratively get batches from a Dataset object
    - Use a sampler object to choose samples
  - **Look at SubsetRandomSampler to split train/val**

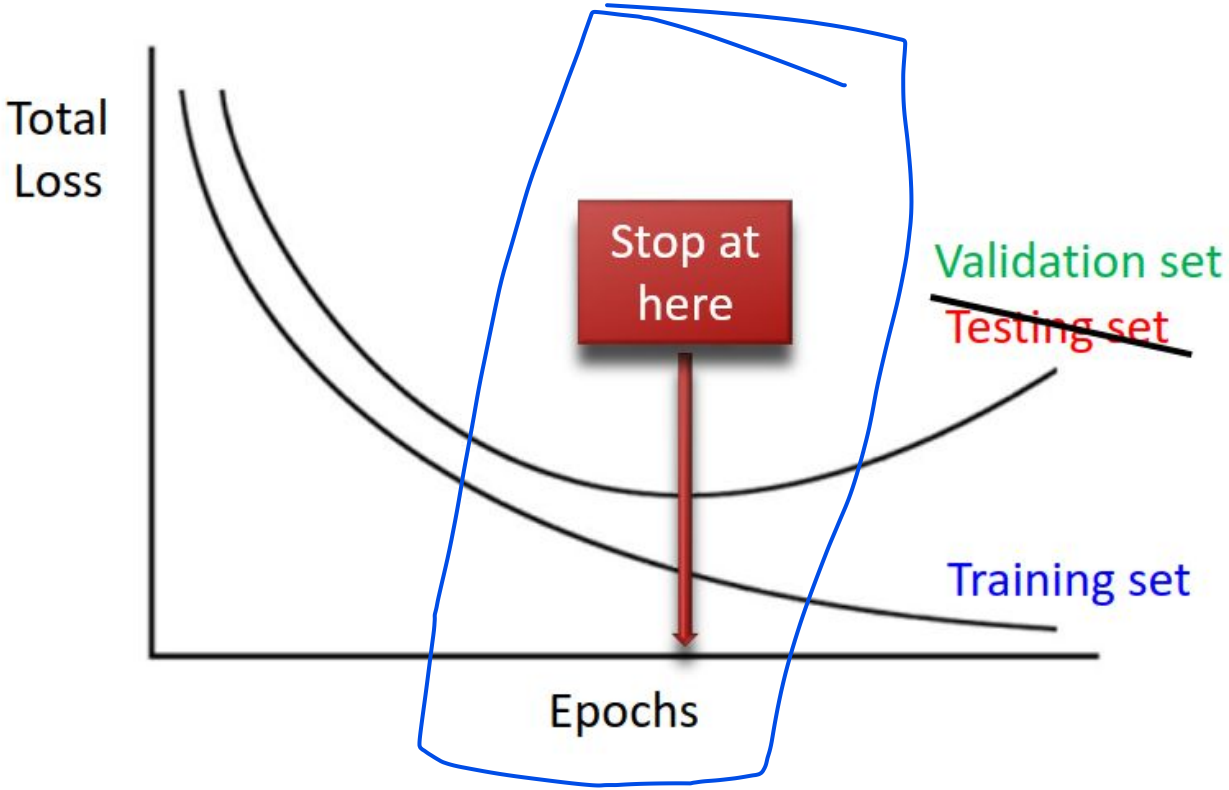
Fundamental problem



babysitting your deep network

1. overfitting and underfitting
2. check accuracy before training [Saxe et al., 2011]
3. Y. Bengio : "*check if the model is powerful enough to overfit, if not then change model structure or make model larger*"

Easy overfitting solution: Early stopping



Normalization

- Normalize/Standardize your dataset
  - Collect mean/std of train set
  - Use it to ensure data is centered with unit std
  - (x-mean)/std

- Batch Normalization
  - Neuron activations can drift deep in the network
  - Normalizing with batch stats

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Possible optimization losses

You should choose the right loss function based on your problem and your data
(here y is the true/expected answer, $f(x)$ the answer predicted by the network).

Classification

- Cross-entropy loss: $L(x) = -(y\ ln(f(x)) + (1-y)ln(1-f(x)))$
- Hinge Loss (max-margin loss, 0-1 loss): $L(x) = max(0, 1-yf(x))$
- …

Regression

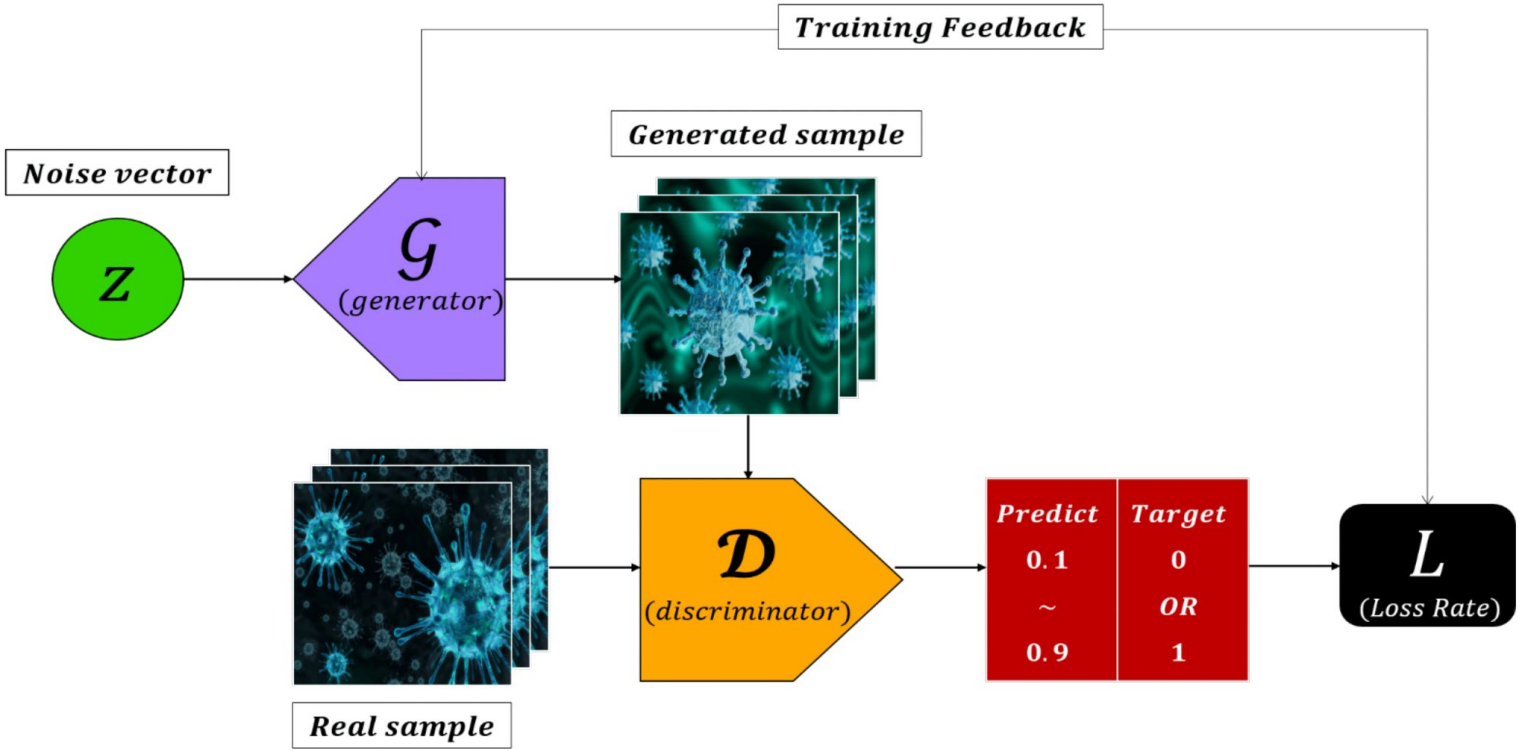
- Mean square loss (or Quadratic Loss): $L(x) = (f(x)-y)^2$
- Mean absolute loss
- …

Retrieval

- Triplet loss
- Cosine similarity, ^{2},…
- …

If the loss is minimized but accuracy is low, you should check the loss function. Maybe it is not the appropriate one for your task.

Adversarial training



So, how do we train better models?

UNIVERSITÉ CÔTE D'AZUR

- Choose suitables losses and scores
- **Use proper activation functions**
- **Look at different initializations**

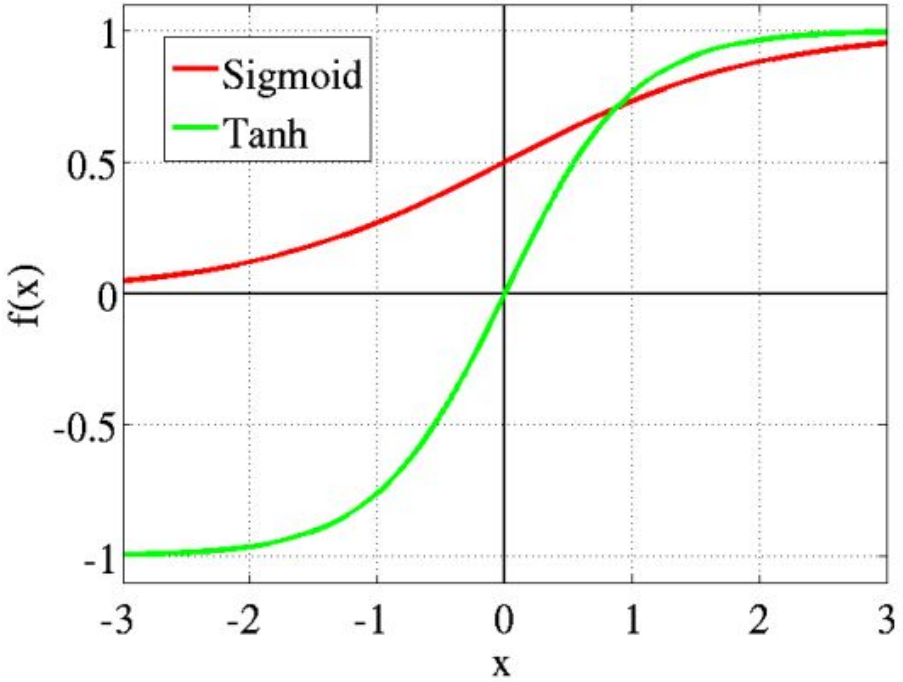
- **Use a good optimizer (SGD variants)**

- **Regularize the model**
- **Augment the dataset**

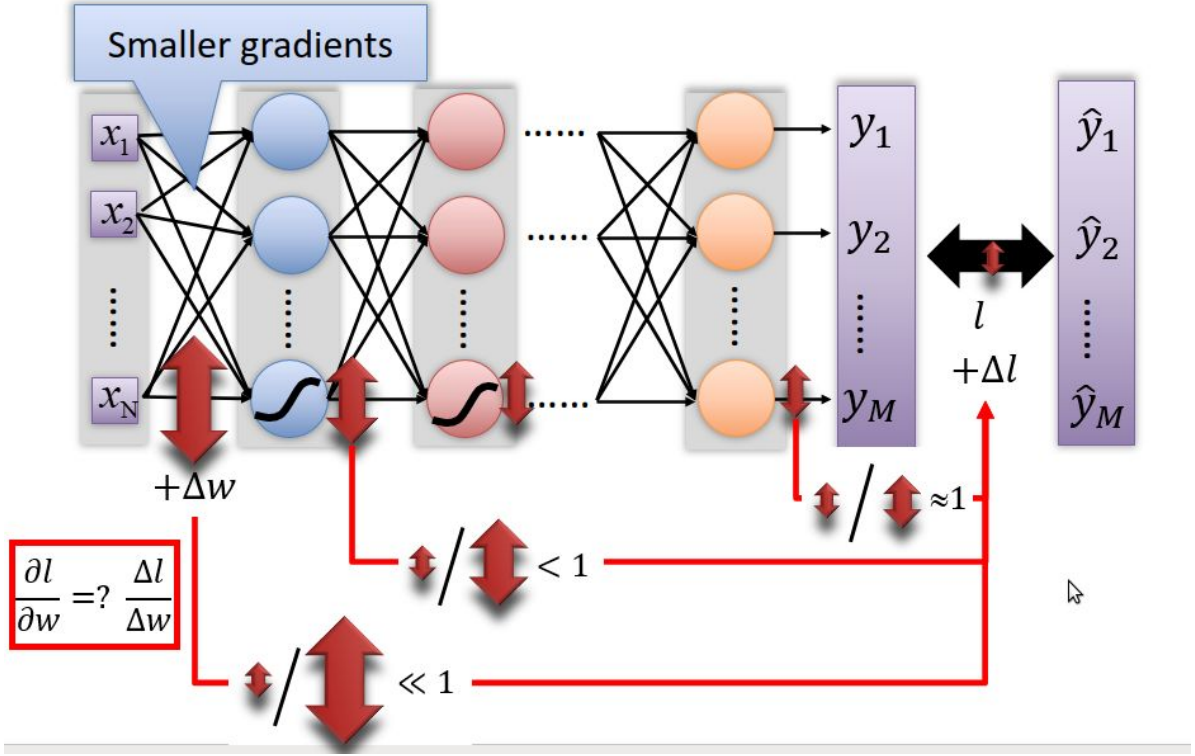
- Pick the right architecture (with the right params)
- Pretrain, transfer, …

Activation toolbox

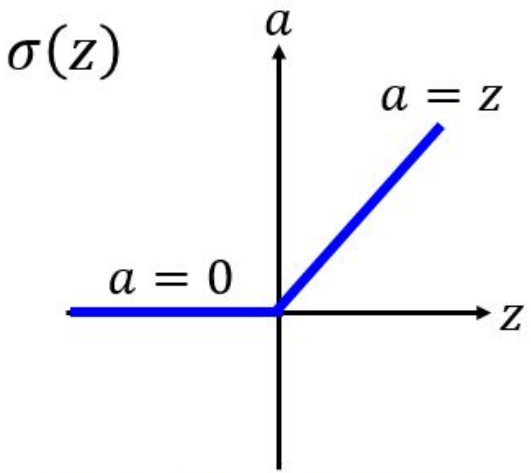
Historical non-linear activations



Vanishing gradient problem



Solution: ReLU

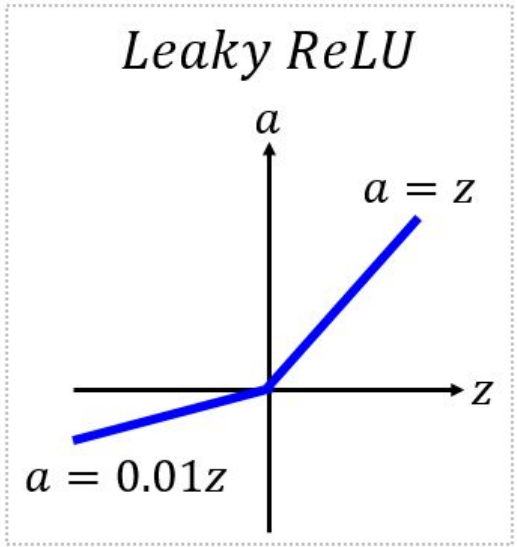


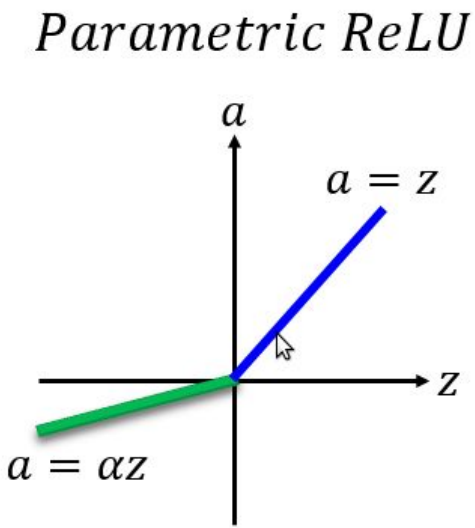
[Xavier Glorot, AISTATS'11]
[Andrew L. Maas, ICML'13]
[Kaiming He, arXiv'15]

Reason:

1. Fast to compute
2. Biological reason
3. Infinite sigmoid with different biases
4. Vanishing gradient problem

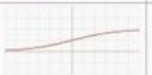
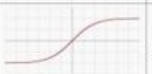
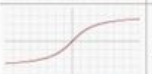
Some ReLU Variants





α also learned by gradient descent

Activation inventory

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Takeaway

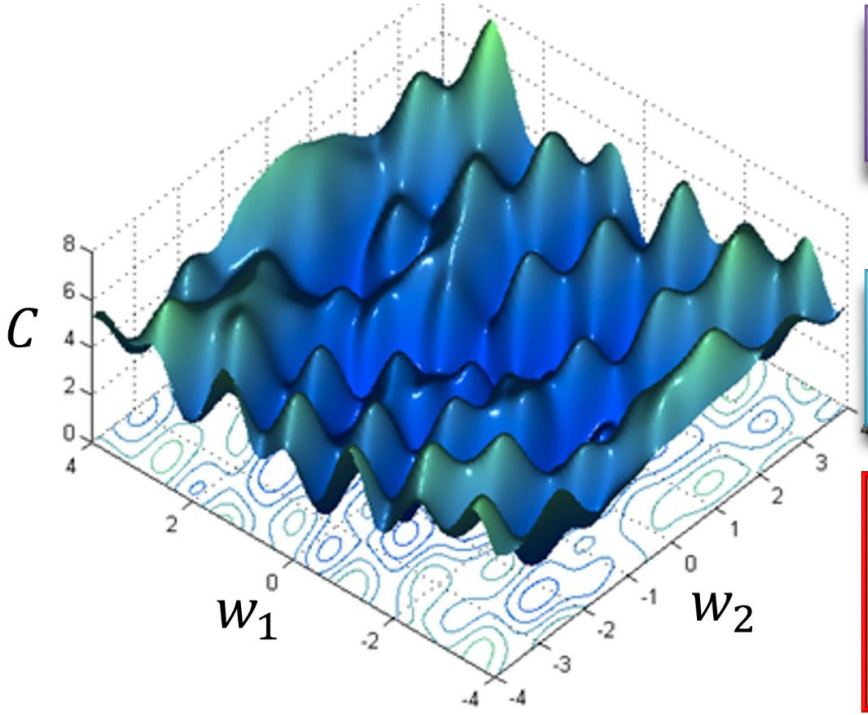
- Activation functions break the linearity
  - We need linear functions for backprop
  - But composition preserves linearity
  - Non-linearity make networks expressive

- Classical activations are cause gradient issues
  - Modern ReLU activations behave better

- **Try different torch.nn activations !**

The initialization problem

Historical non-linear activations



Different initial point W^0

Reach different minima, thus different results

The Loss Surfaces of Multilayer Networks (AISTAT 2015)
A. Choromanska, M. Henaff, M. Mathieu, G. Ben Arous, Y. LeCun

Historical non-linear activations

UNIVERSITÉ CÔTE D'AZUR

How to initialize the weights?

- ***All zero initialization (W=0)**: not good when the network **is deep**, every neuron computes the same output, have same gradients during back-propagation*

- *Zero-mean Gaussian with 0.01 stddev, (W~N (0, 0.01))*
- *Uniform Distribution*
- *Orthogonal: The Eigen values of an orthogonally initialized matrix are one.*

- ***Glorot normal/uniform** (aka **Xavier normal/uniform**): zero-mean gaussian initialization with variance scaled by number of input neurons + number of output neurons (Glorot et al., 2010), keeping the signal in a reasonable range of values through many layers*
  (W ~ $\mathcal{N}$ (0, 2/(input_neurons+output_neurons)))

Tanh!

- ***He normal/uniform**: zero-mean gaussian initialization scaled by number of input neurons*
  (W ~ $\mathcal{N}$ (0, 1/input_neurons))

ReLU Family

Historical non-linear activations

UNIVERSITÉ CÔTE D'AZUR

How to initialize the weights?

- ***All zero initialization (W=0)**: not good when the network **is deep**, every neuron computes the same output, have same gradients during back-propagation*

- *Zero-mean Gaussian with 0.01 stddev, (W~N (0, 0.01))*
- *Uniform Distribution*
- *Orthogonal: The Eigen values of an orthogonally initialized matrix are one.*

Less important with BN!

- ***Glorot normal/uniform** (aka **Xavier normal/uniform**): zero-mean gaussian initialization with variance scaled by number of input neurons + number of output neurons (Glorot et al., 2010), keeping the signal in a reasonable range of values through many layers*
  (W ~ $\mathcal{N}$ (0, 2/(input_neurons+output_neurons)))

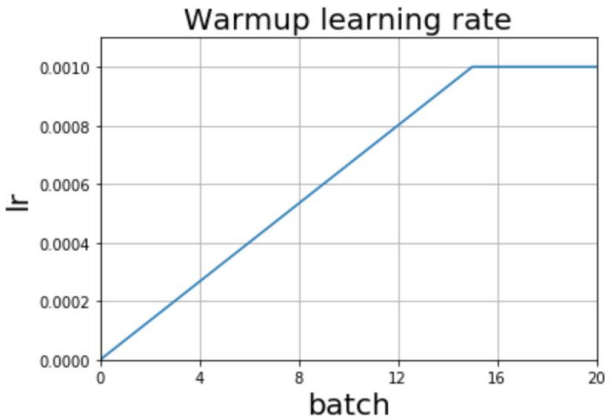
Tanh!

- ***He normal/uniform**: zero-mean gaussian initialization scaled by number of input neurons*
  (W ~ $\mathcal{N}$ (0, 1/input_neurons))

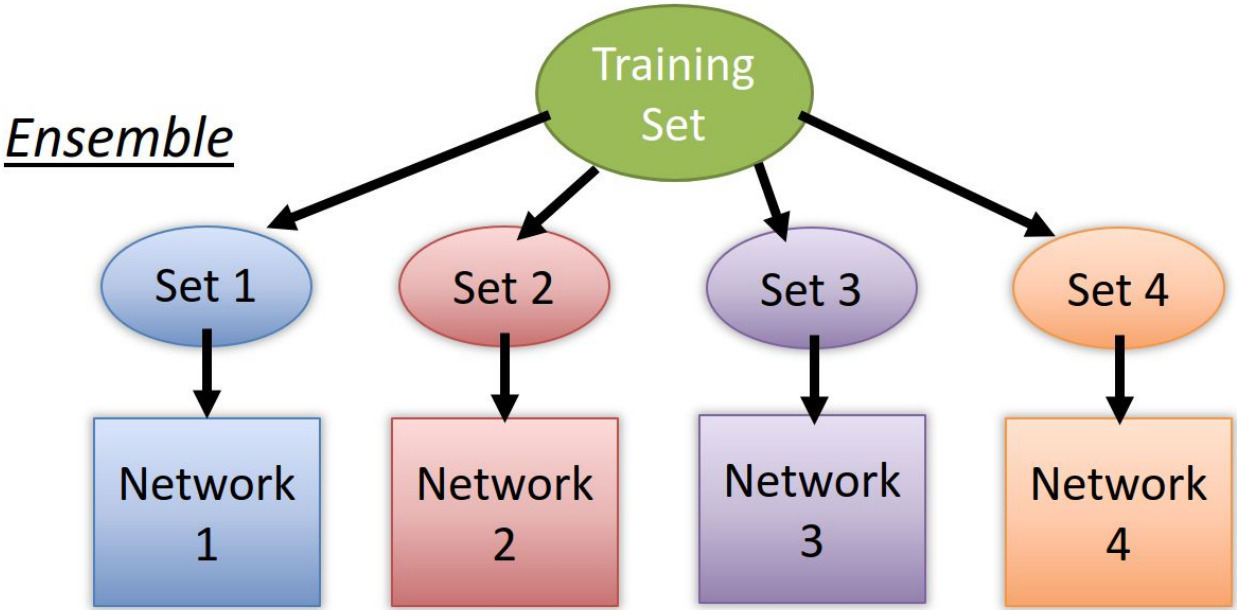
ReLU Family

Warmup learning rate

- Start with very small learning rate
- Plays on initialization
  - “Aligns” weights to problem
  - **Then** do actual training



Ensembling

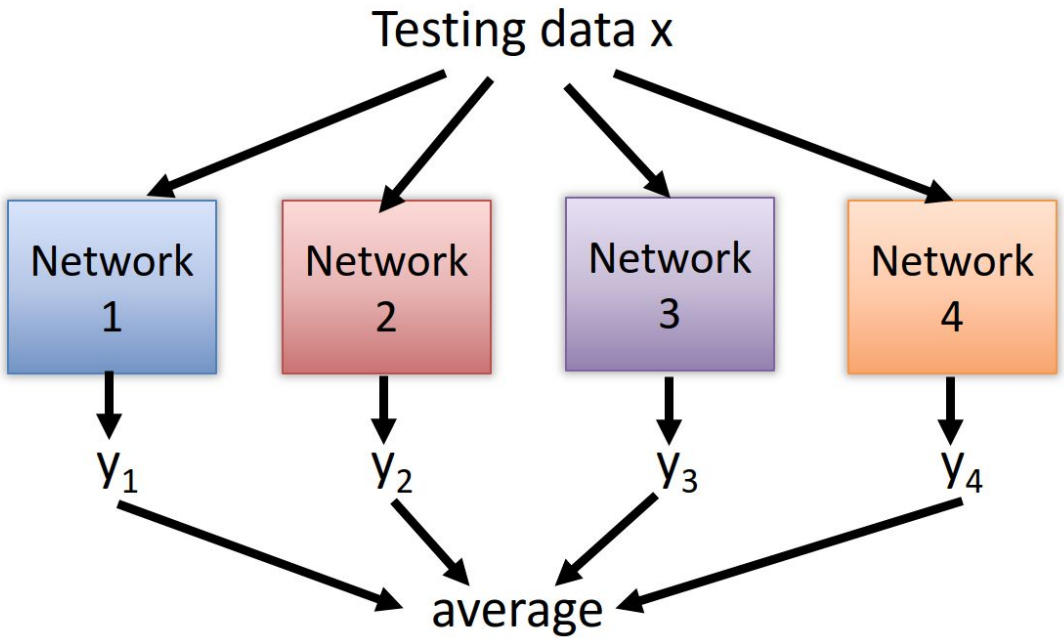


Train a bunch of networks with different structures

Ensembling (explicit)

UNIVERSITÉ CÔTE D'AZUR

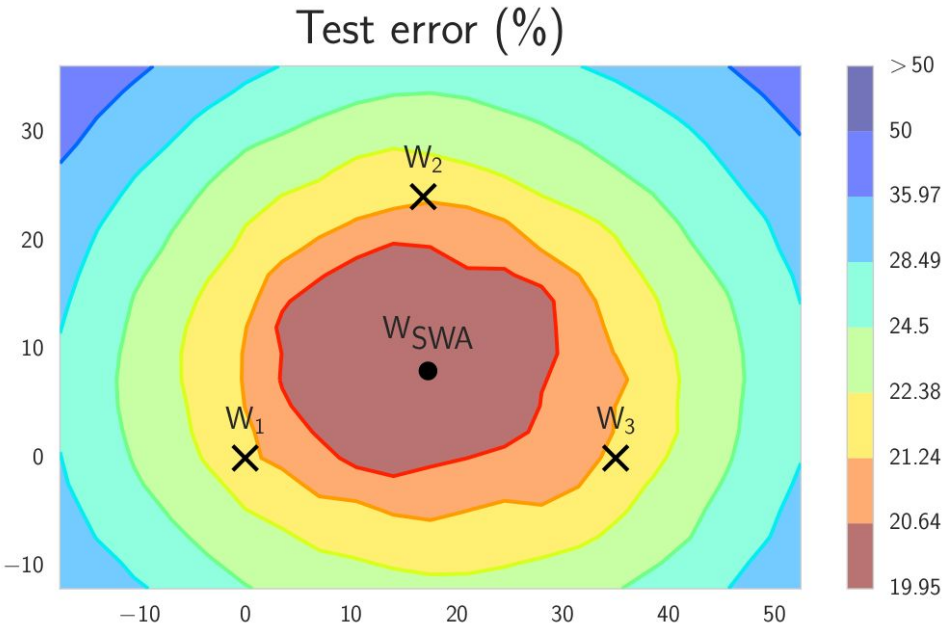
Ensemble



Ensembling (implicit) via weight averaging

UNIVERSITÉ CÔTE D'AZUR

- Can average weights
  - Over time (exponential moving average)
  - Of different models fine-tuned from a root model
- Implicit ensembling
  - Poorly understood

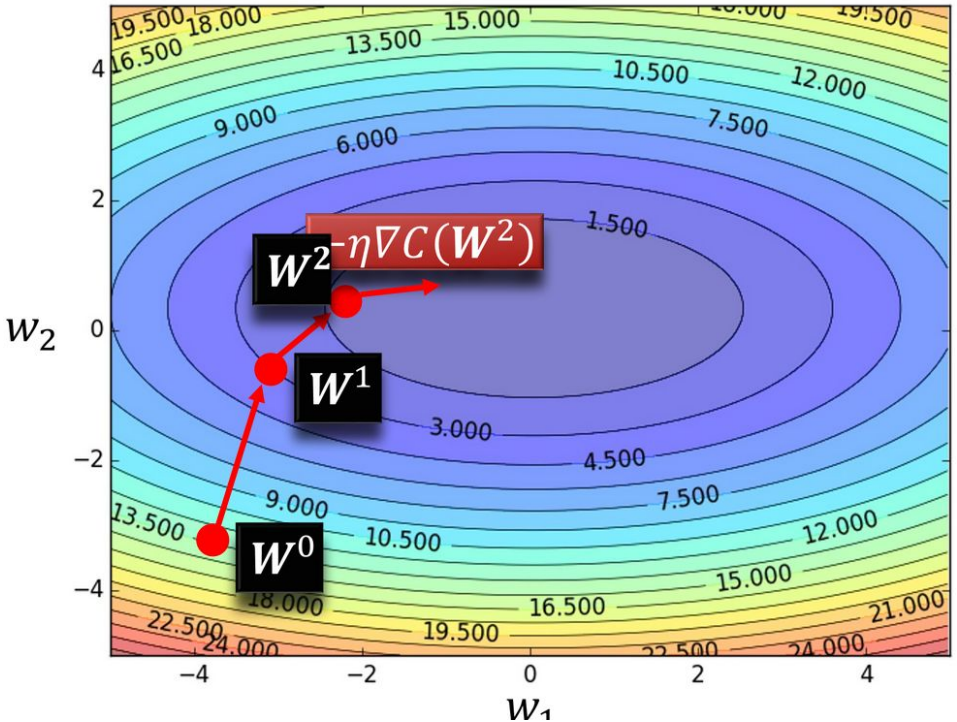


Takeaway

- Very dependent on network initialization
  - Good init -> good results
  - Bad init -> Bad results
- Known schemes to guarantee standardized properties of the outputs
- Can ensemble decisions from multiple initializations of the model
- **Look at weight values after initialization**

Gradients and optimizers

Remember SGD?



Randomly pick a starting point $\boldsymbol{W}^0$

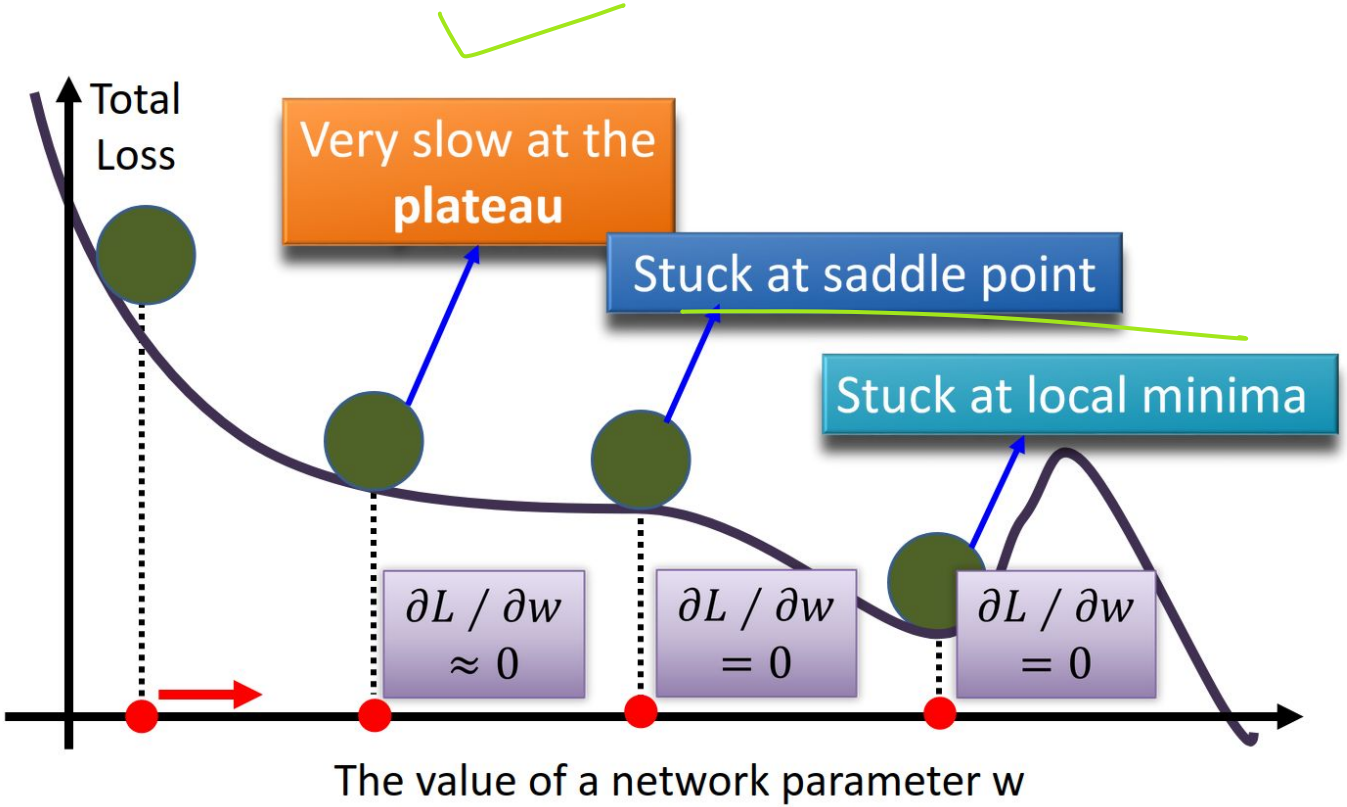
Compute the negative gradient at $\boldsymbol{W}^0$

$\Rightarrow -\nabla C(\boldsymbol{W}^0)$

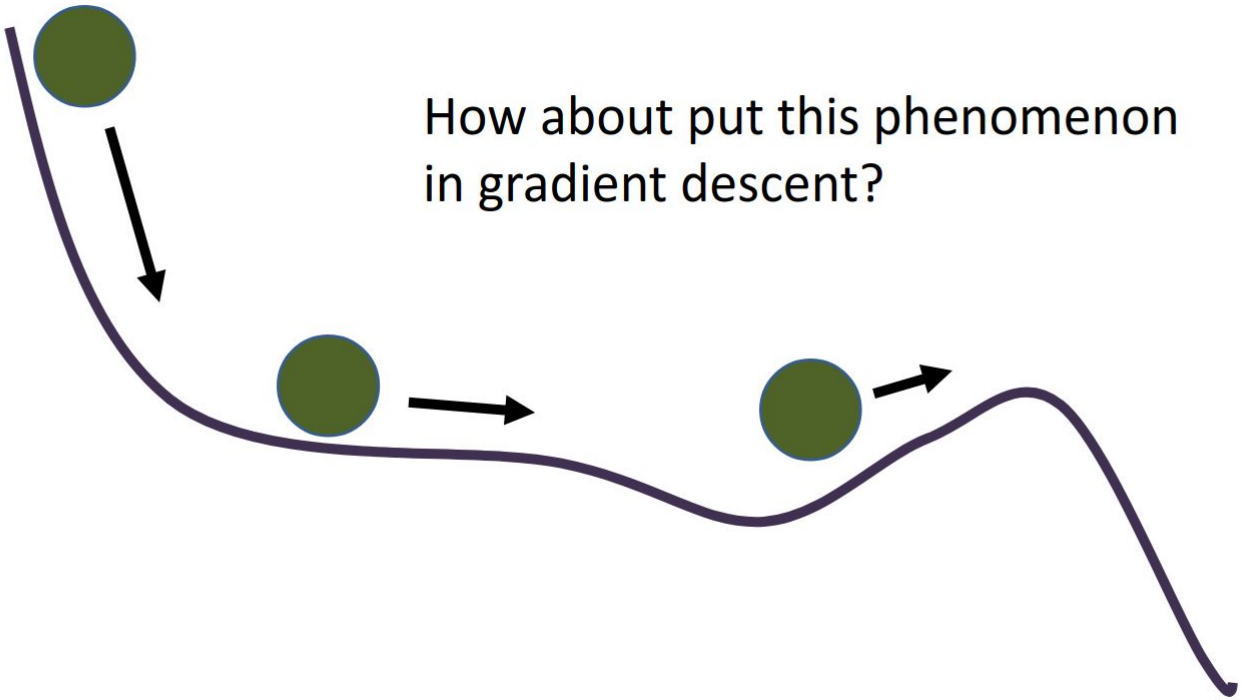
Times the learning rate η

$\Rightarrow -\eta\nabla C(\boldsymbol{W}^0)$

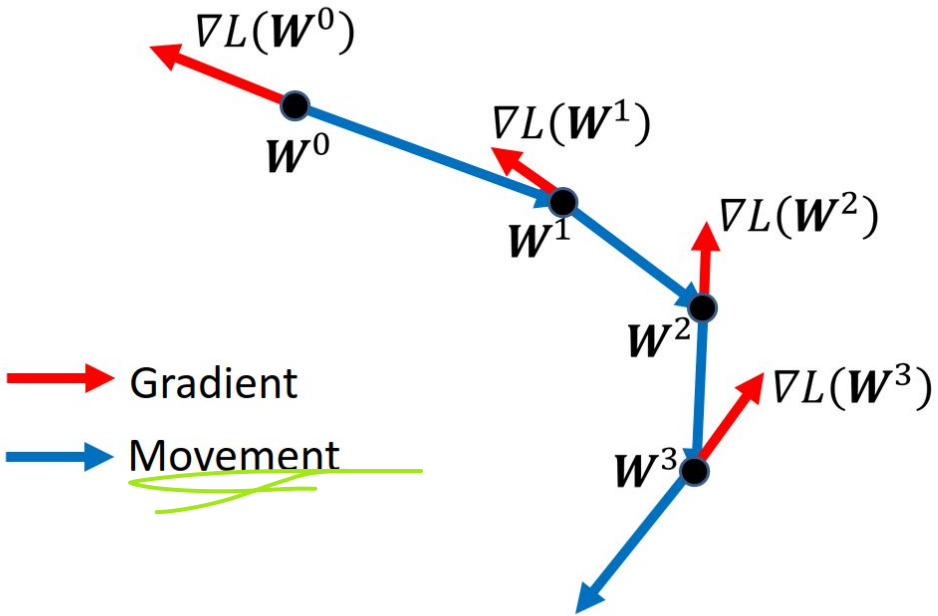
What is happening in a NN



Cannot we build "momentum" as go?



Vanilla SGD



Start at position $\boldsymbol{W}^0$

Compute gradient at $\boldsymbol{W}^0$

Move to $\boldsymbol{W}^1 = \boldsymbol{W}^0 - \eta\nabla L(\boldsymbol{W}^0)$

Compute gradient at $\boldsymbol{W}^1$

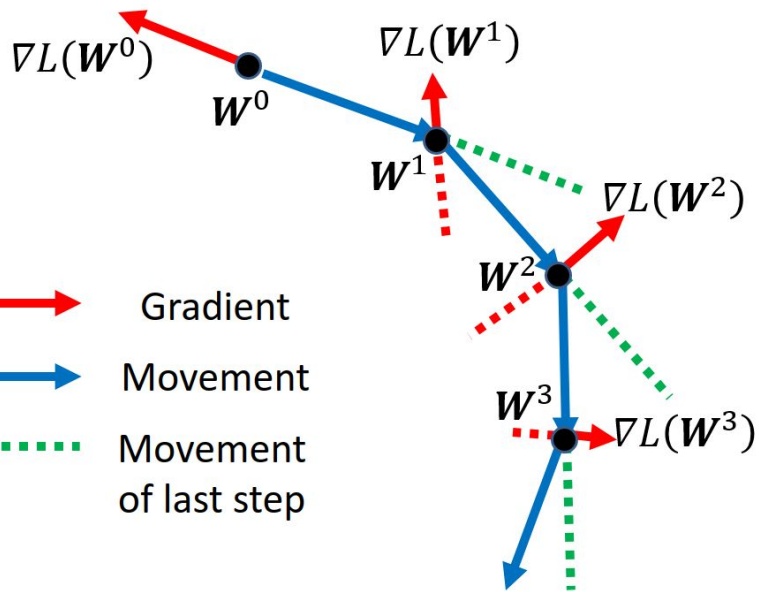
Move to $\boldsymbol{W}^2 = \boldsymbol{W}^1 - \eta\nabla L(\boldsymbol{W}^1)$

⋮

Stop until $\nabla L(\boldsymbol{W}^t) \approx 0$

Momentum SGD

Movement: movement of last step minus gradient at present



Start at point $\boldsymbol{W}^0$

Movement $v^0=0$

Compute gradient at $\boldsymbol{W}^0$

Movement $v^1 = \lambda v^0 - \eta \nabla L(\boldsymbol{W}^0)$

Move to $\boldsymbol{W}^1 = \boldsymbol{W}^0 + v^1$

Compute gradient at $\boldsymbol{W}^1$

Movement $v^2 = \lambda v^1 - \eta \nabla L(\boldsymbol{W}^1)$

Move to $\boldsymbol{W}^2 = \boldsymbol{W}^1 + v^2$

Movement not just based on gradient, but previous movement.

Momentum SGD

Movement: movement of last step minus gradient at present

v^i is actually the weighted sum of all the previous gradient: $\nabla L(\boldsymbol{W}^0), \nabla L(\boldsymbol{W}^1), \ldots \nabla L(\boldsymbol{W}^{i-1})$

$v^0 = 0$

$v^1 = -\eta \nabla L(\boldsymbol{W}^0)$

$v^2 = -\lambda \eta \nabla L(\boldsymbol{W}^0) - \eta \nabla L(\boldsymbol{W}^1)$

⋮

Start at point $\boldsymbol{W}^0$

Movement $v^0=0$

Compute gradient at $\boldsymbol{W}^0$

Movement $v^1 = \lambda v^0 - \eta \nabla L(\boldsymbol{W}^0)$

Move to $\boldsymbol{W}^1 = \boldsymbol{W}^0 + v^1$

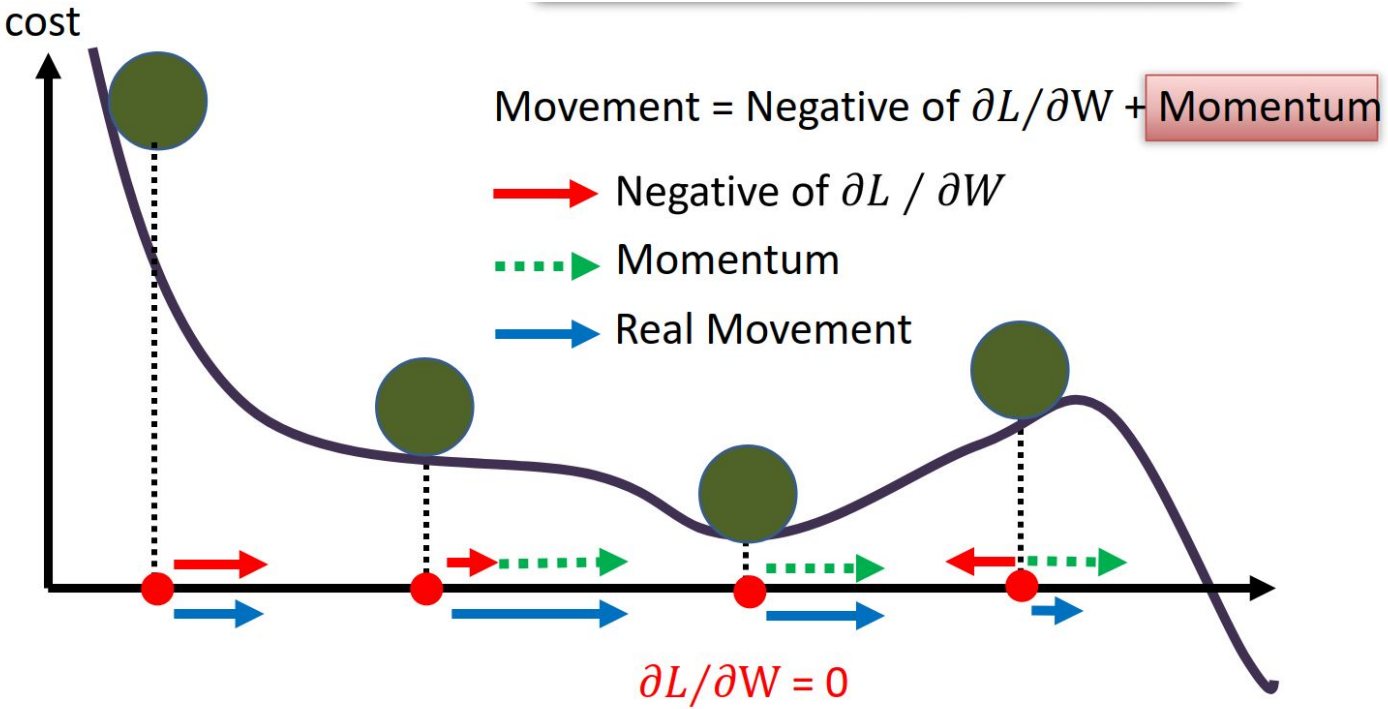
Compute gradient at $\boldsymbol{W}^1$

Movement $v^2 = \lambda v^1 - \eta \nabla L(\boldsymbol{W}^1)$

Move to $\boldsymbol{W}^2 = \boldsymbol{W}^1 + v^2$

Movement not just based on gradient, but previous movement

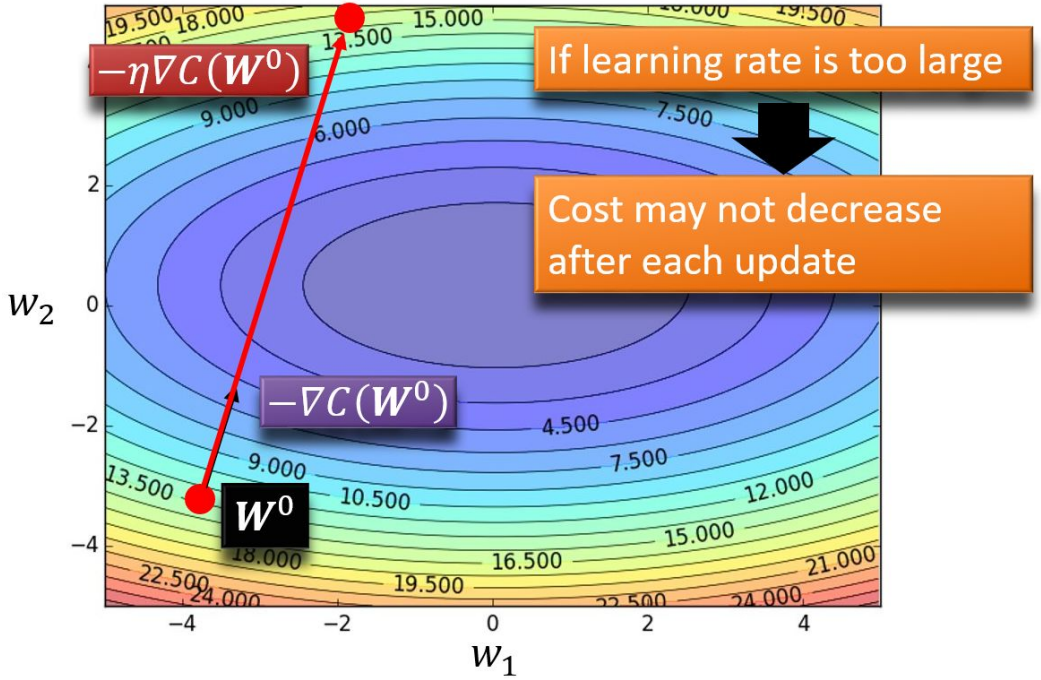
Momentum SGD



Adapting the learning rate

- ***Adaptive Learning Rate***

Set the learning rate η carefully



Adapting the learning rate

UNIVERSITÉ CÔTE D'AZUR

- Popular & Simple Idea: Reduce the learning rate by some factor every few epochs.
  - At the beginning, we are far from the destination, so we use larger learning rate
  - After several epochs, we are close to the destination, so we reduce the learning rate
  - E.g. 1/t decay: $\eta^t = \eta/\sqrt{t+1}$
- Learning rate cannot be one-size-fits-all
  - Giving different parameters different learning rates

Adapting the learning rate

Original Gradient Descent

$$\boldsymbol{W}^t \leftarrow \boldsymbol{W}^{t-1} - \eta \nabla C(\boldsymbol{W}^{t-1})$$

Each parameter w are considered separately

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \eta_{\boldsymbol{W}} g^t \qquad g^t = \frac{\partial C(\boldsymbol{W}^t)}{\partial \boldsymbol{W}}$$

Parameter dependent learning rate

$$\eta_{\boldsymbol{W}} = \frac{\eta}{\sqrt{\sum_{i=0}^{t} (g^i)^2}}$$

constant

g^i is $\partial L / \partial w$ obtained at the i-th update

Summation of the square of the previous derivatives

Adapting the learning rate

UNIVERSITÉ CÔTE D'AZUR

$$\eta_w = \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}}$$

- ***Adagrad***

	g^0
w_1	0.1

	g^0
w_2	20.0

Learning rate:

$$\frac{\eta}{\sqrt{0.1^2}} = \frac{\eta}{0.1} \quad \leftrightarrow \quad \frac{\eta}{\sqrt{20^2}} = \frac{\eta}{20}$$

$$\frac{\eta}{\sqrt{0.1^2 + 0.2^2}} = \frac{\eta}{0.22} \quad \leftrightarrow \quad \frac{\eta}{\sqrt{20^2 + 10^2}} = \frac{\eta}{22}$$

Observation: 1. Learning rate is smaller and smaller for all parameters

2. Smaller derivatives, larger learning rate, and vice versa

RMSProp

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2}$$

$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

Root Mean Square of the gradients with previous gradients being decayed

Adam: Momentum + RMSProp

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : *Initial Learning rate*

g_t : *Gradient at time t along* ω^j

ν_t : *Exponential Average of gradients along* ω_j

s_t : *Exponential Average of squares of gradients along* ω_j

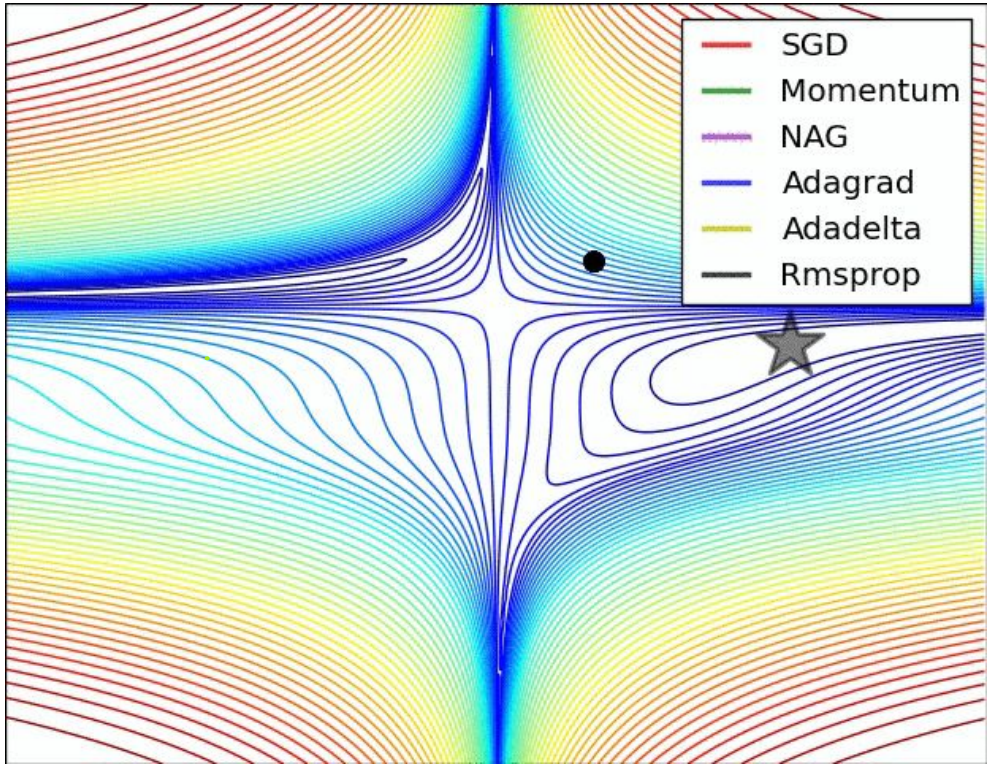
β_1, β_2 : *Hyperparameters*

Optimizer toolbox

UNIVERSITÉ CÔTE D'AZUR

- Adagrad [John Duchi, JMLR'11]
- RMSprop
  - https://www.youtube.com/watch?v=O3sxAc4hxZU
- Adadelta [Matthew D. Zeiler, arXiv'12]
- Adam [Diederik P. Kingma, ICLR'15] = RMSprop + Momentum
- AdaSecant [Caglar Gulcehre, arXiv'14]
- "No more pesky learning rates" [Tom Schaul, arXiv'12]
- Nadam
  - http://cs229.stanford.edu/proj2015/054_report.pdf

Visualization



A word on learning rate and batch size

UNIVERSITÉ CÔTE D'AZUR

- Small batch = Good generalization
  - Because of noisiness
    - Kind of but not really

- Perfectly possible to use large batches
  - Need to scale learning rate
  - If SGD
    - xN batch -> x N learning rate
  - If adaptive (Adam, …)
    - Less clear, xN B -> x N or x sqrt(N) lr
    - Linked to Hessian eigenvalues

Really big batch sizes

UNIVERSITÉ CÔTE D'AZUR

- Scaling laws break down at very large batch sizes and learning rates
  - Fatal issues with gradient noise
- Solution: account for gradient quality or “trust” in LR

$$\lambda^l = \eta \times \frac{||w^l||}{||\nabla L(w^l)||}$$

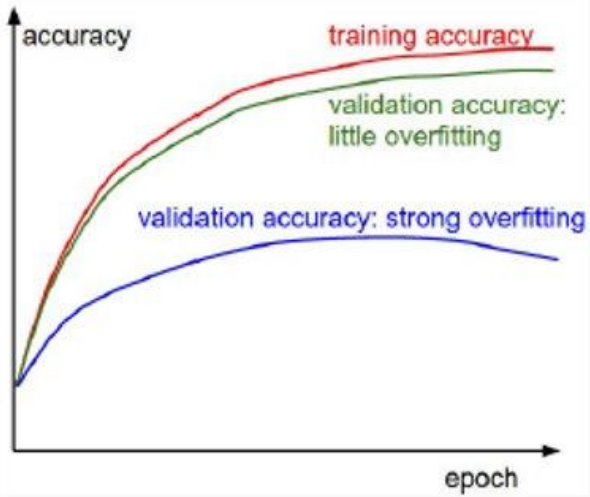
- Adaptable to classical optimizers
  - SGD -> LARS
  - Adam -> LAMB

Takeaway

- Momentum can help get over small “hills”
  - Can avoid very shallow minima
- Adapt the learning rate (Step, cosine, exp, …)
  - Get smaller learning rates to go further “into” the minimum
- Per parameter adaptations (Adam, …)
  - Move differently for different parameters
- **Look at the torch.optim optimizer**

Model regularization

Overfitting



babysitting your deep network

1. overfitting and underfitting
2. check accuracy before training [Saxe et al., 2011]
3. Y. Bengio : "*check if the model is powerful enough to overfit, if not then change model structure or make model larger*"

Weight regularization

- New loss function to be minimized
  - Find a set of weight not only minimizing original cost but also close to zero

$$L'(W) = L(W) + \lambda \frac{1}{2} \|W\|_2$$ → Regularization term

$$W = \{w_1, w_2, \ldots\}$$

L2 regularization also known as ridge, or weight decay
(***The most widely used regularization method***):

$$\|W\|_2 = (w_1)^2 + (w_2)^2 + \ldots$$

(usually not consider biases)

Weight regularization

$$\mathbf{L}'(W) = L(W) + \lambda \frac{1}{2}\|W\|_2 \quad \text{Gradient:} \quad \frac{\partial \mathrm{L}'}{\partial w} = \frac{\partial \mathrm{L}}{\partial w} + \lambda w$$

Update:

$$w^{t+1} \rightarrow w^t - \eta \frac{\partial \mathrm{L}'}{\partial w} = w^t - \eta\left(\frac{\partial \mathrm{L}}{\partial w} + \lambda w^t\right)$$

$$= (1 - \eta\lambda) w^t - \eta \frac{\partial \mathrm{L}}{\partial w}$$

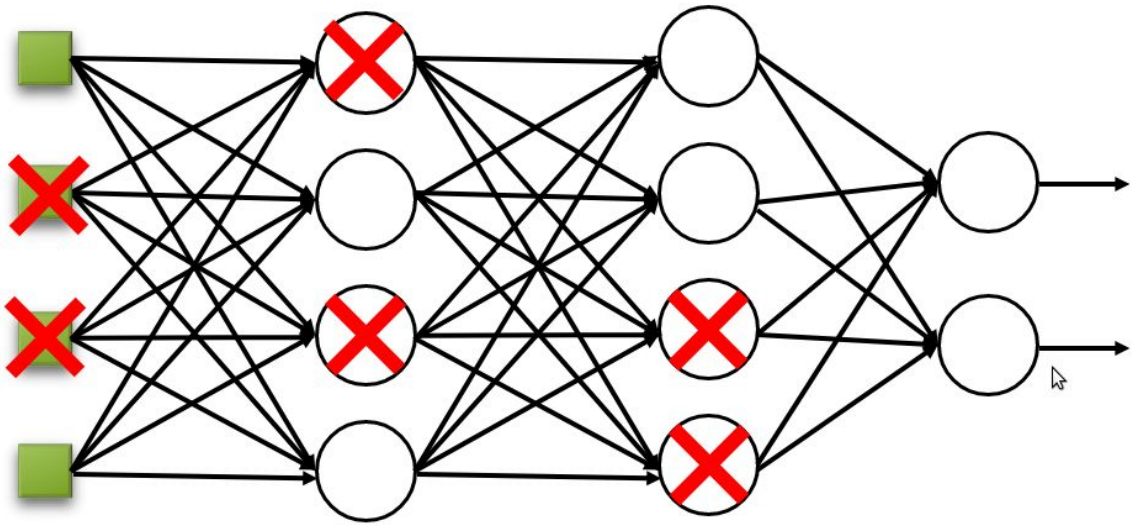
Weight Decay

Closer to zero

Can use other norms: L1, L2, … and penalize other things (gradients, …)

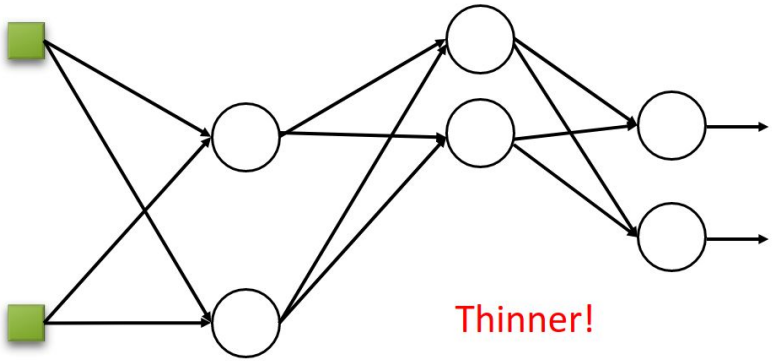
Dropout

Training:



Dropout

Training:

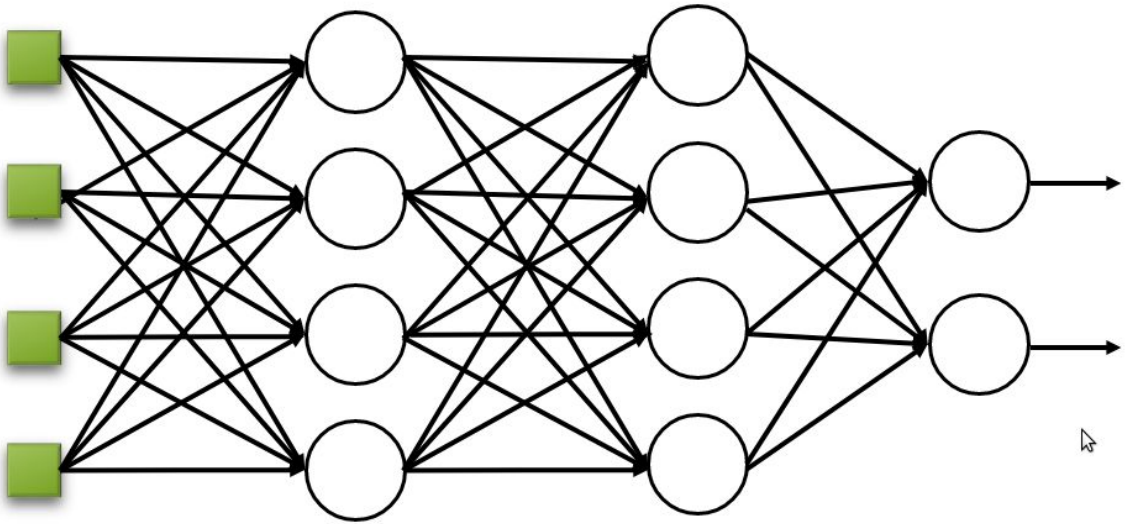


- **Each time before updating the parameters**
  - Each neuron has p% to dropout
    - **The structure of the network is changed.**
  - Using the new network for training

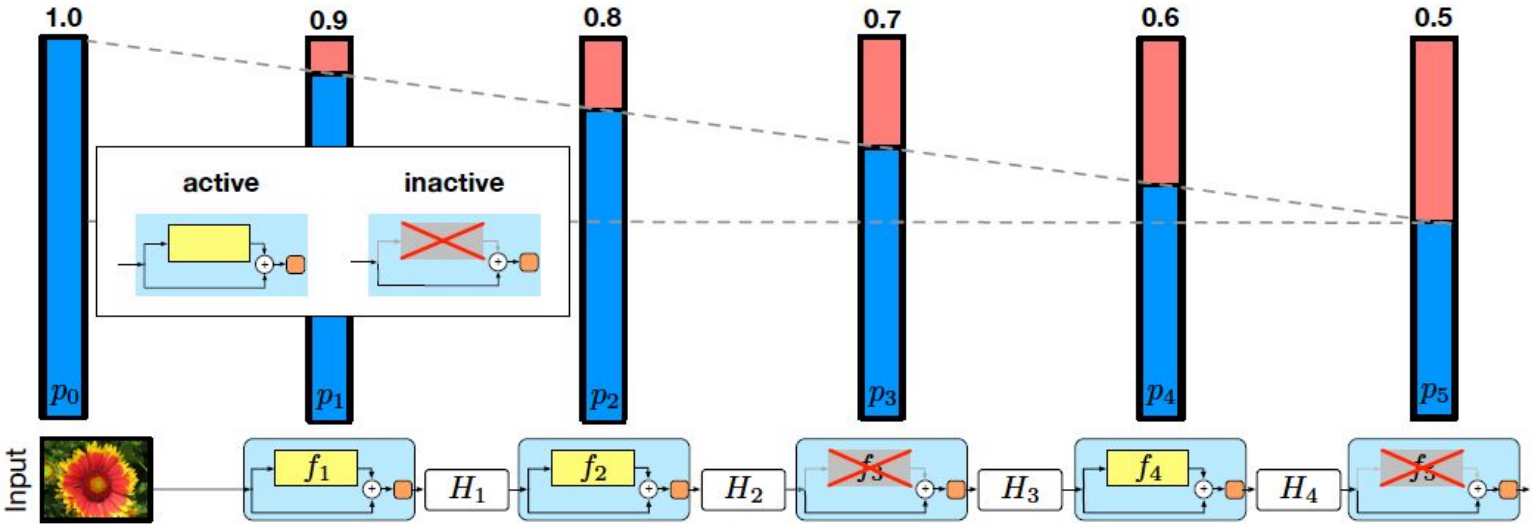
For each mini-batch, we resample the dropout neurons

Dropout

Testing:

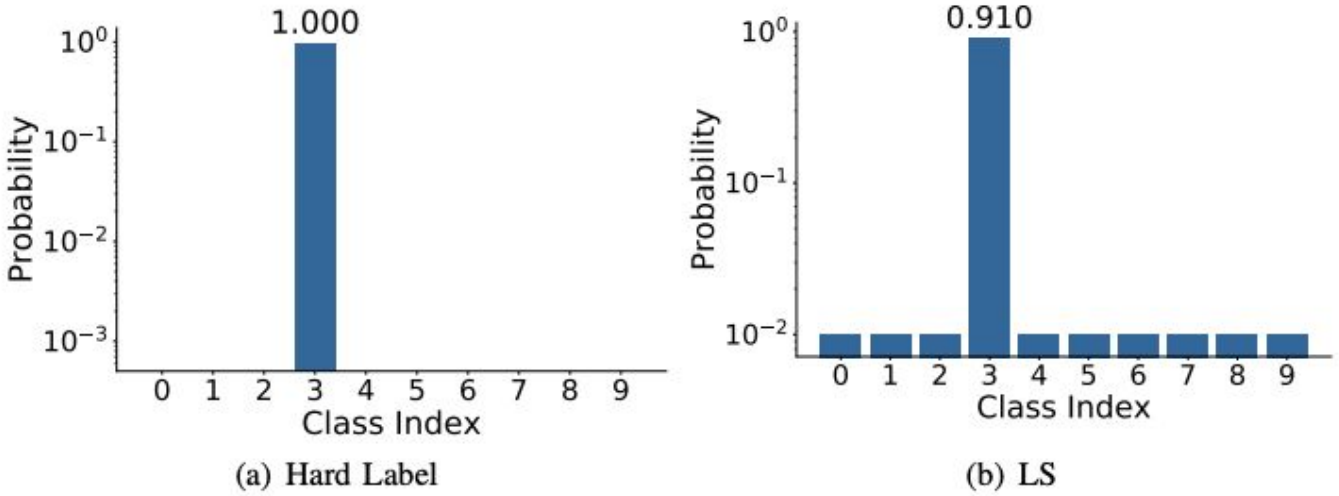


Stochastic depth



Label smoothing

- Make the model less confident
  - Reduce the predicted confidence
  - Modern models = overconfident



Takeaway

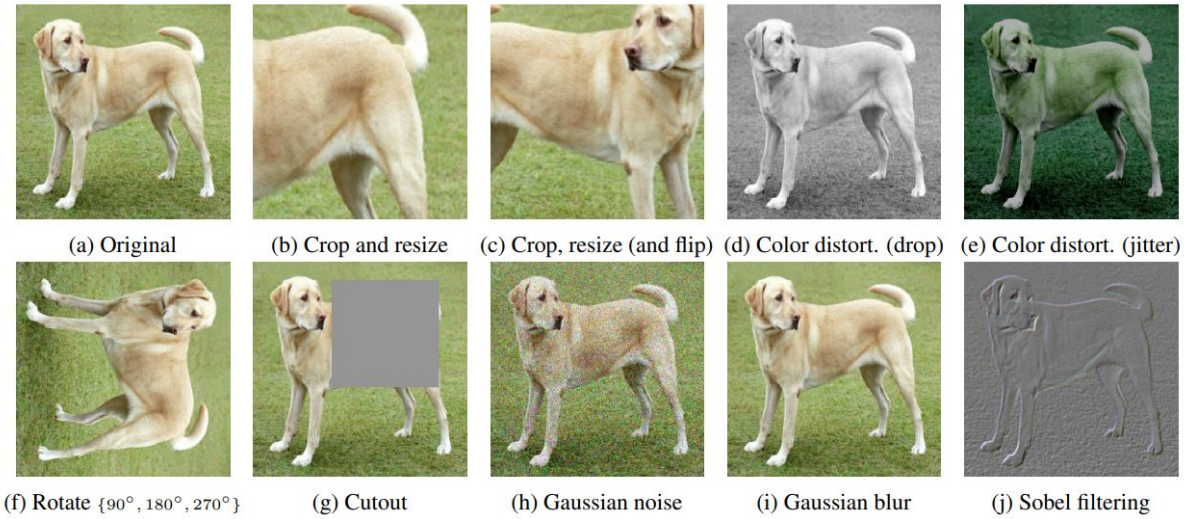
- Very powerful networks can overfit
  - Memorize the training set
  - But no useful conclusions for generalizations

- Try to force the model not to use its full capacity
  - Regularization on the weights
  - Or other things if you want!

- **Torch optimizers have weight decay parameter, but you can also do it by hand!**

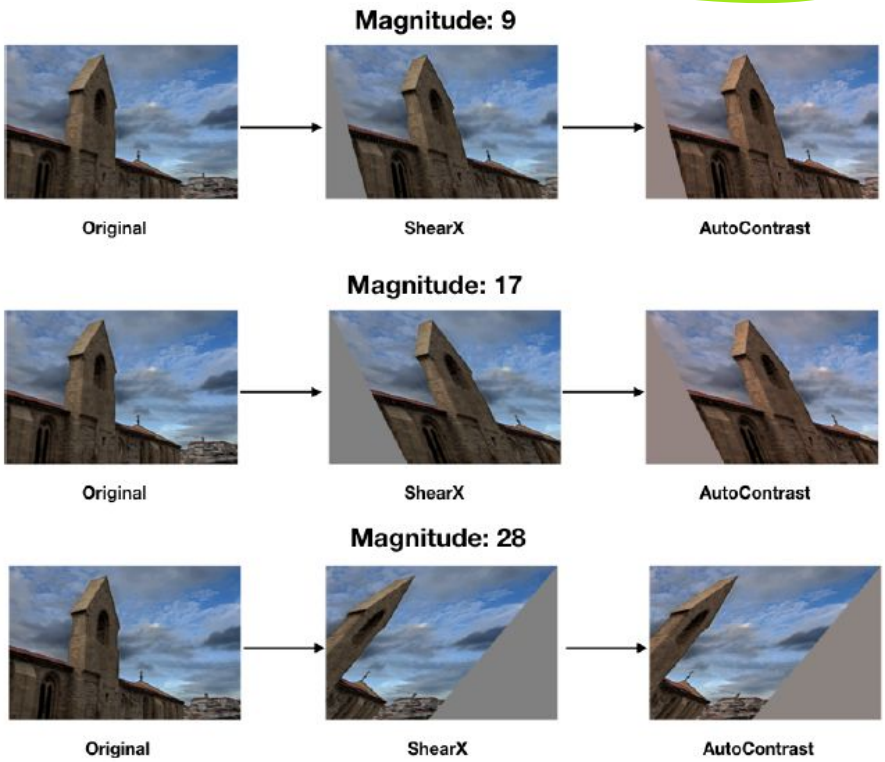
Data Augmentation

Standard data augmentation

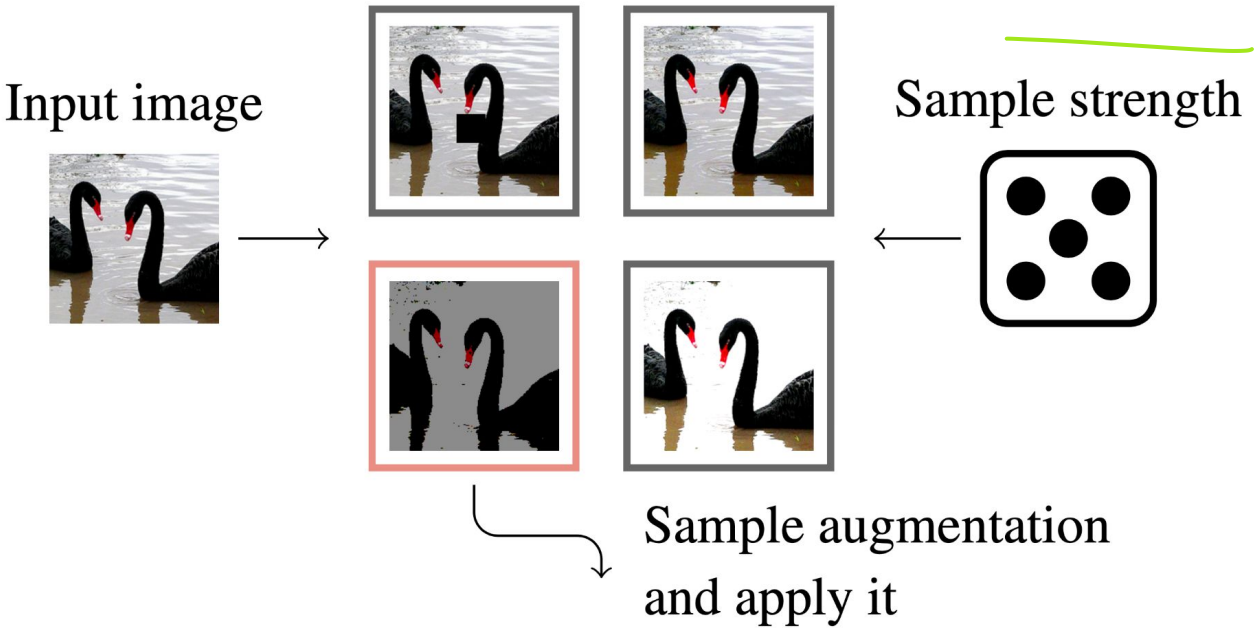


- Dataset can be too small and cause overfitting
  - Make more samples
    - “New” images, same label
  - Problem specific

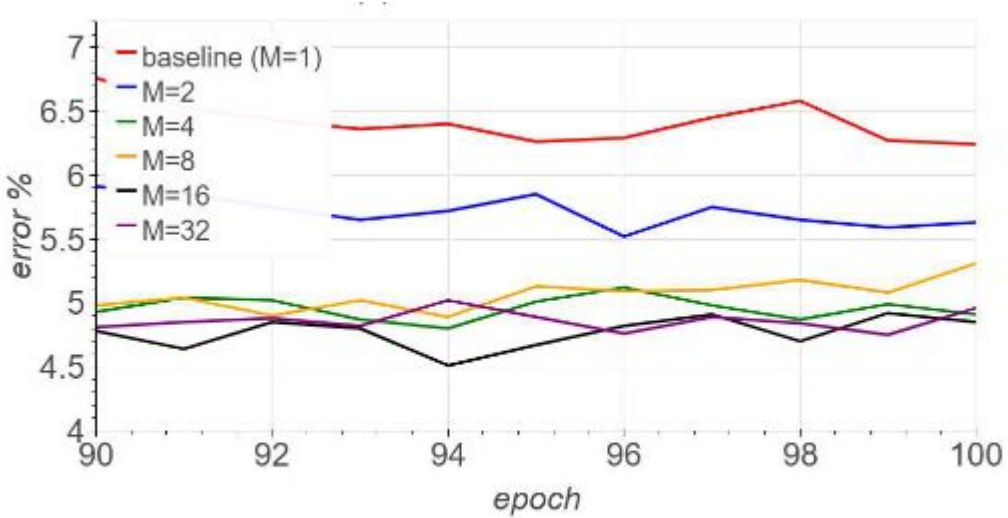
Strong augmentation: RandAugment



Strong augmentation: TrivialAugment



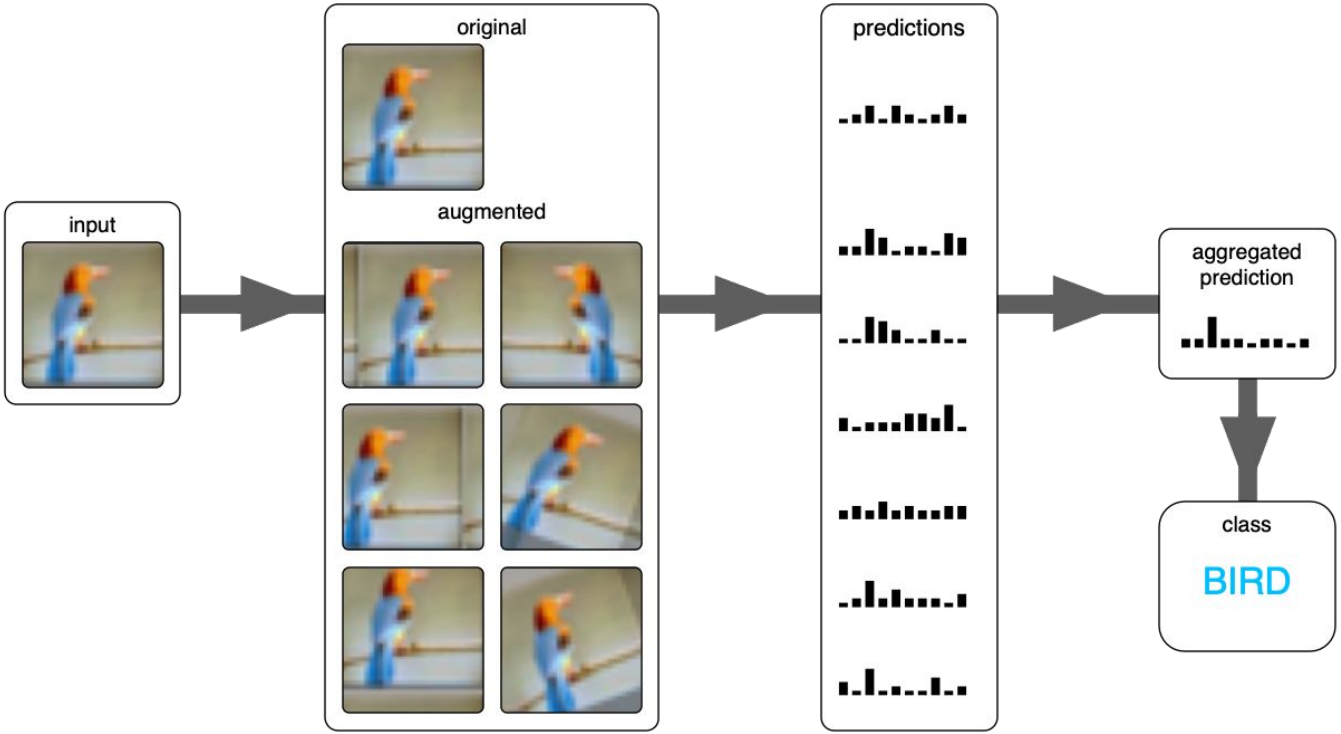
Repeated augmentations



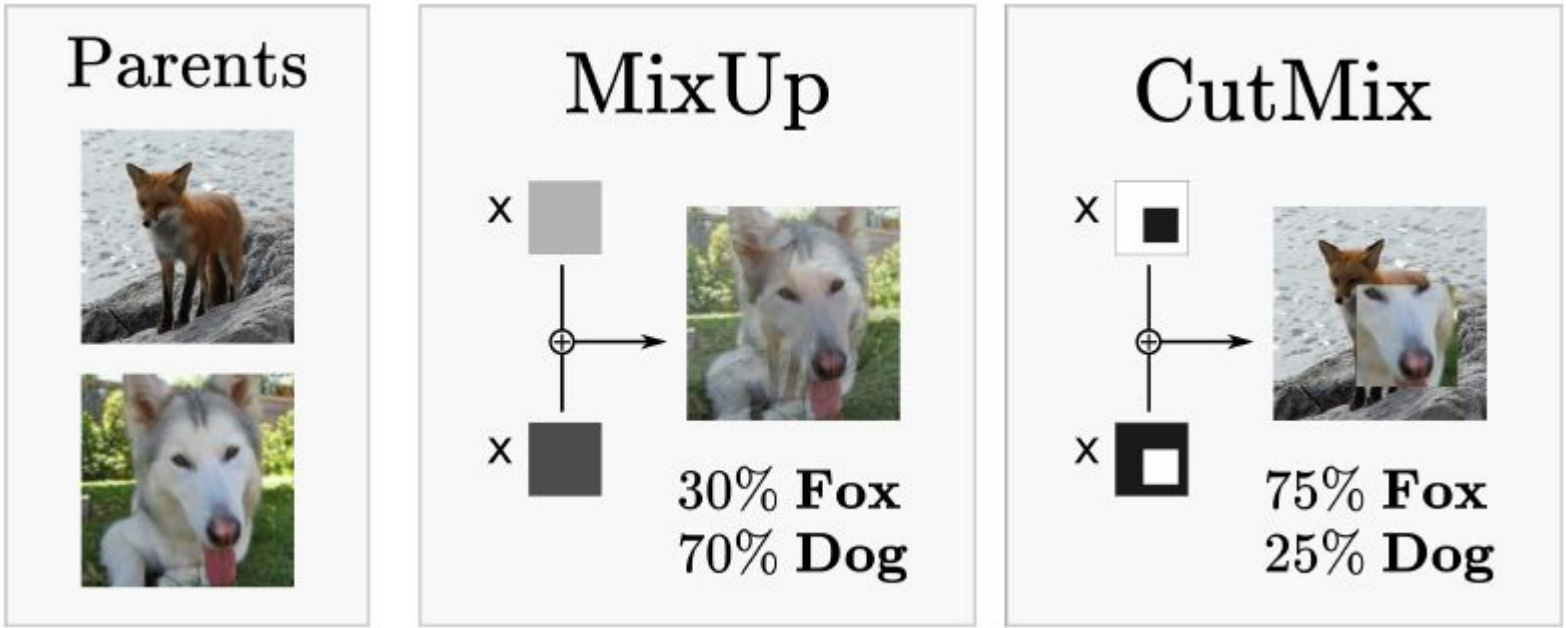
(b) Final Validation error

- Show the same image augmented different ways!

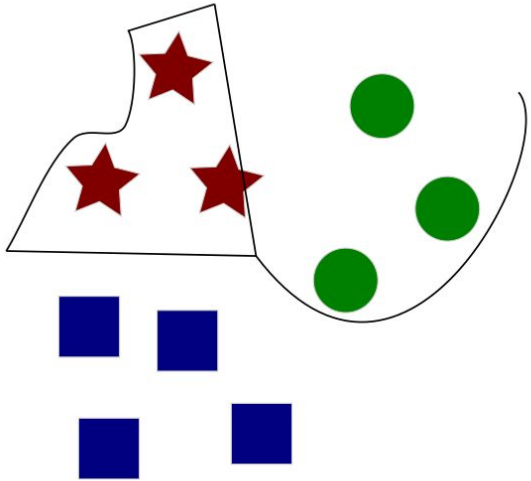
Test time augmentation



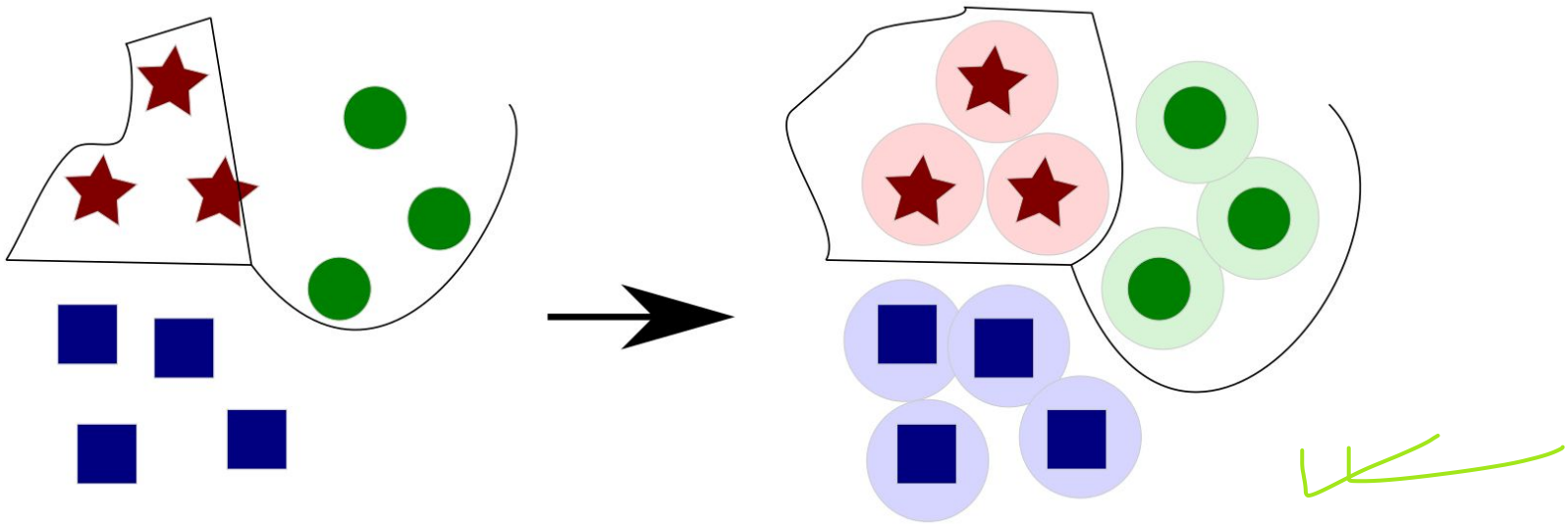
Mixing Data Augmentation



Role of Data augmentation

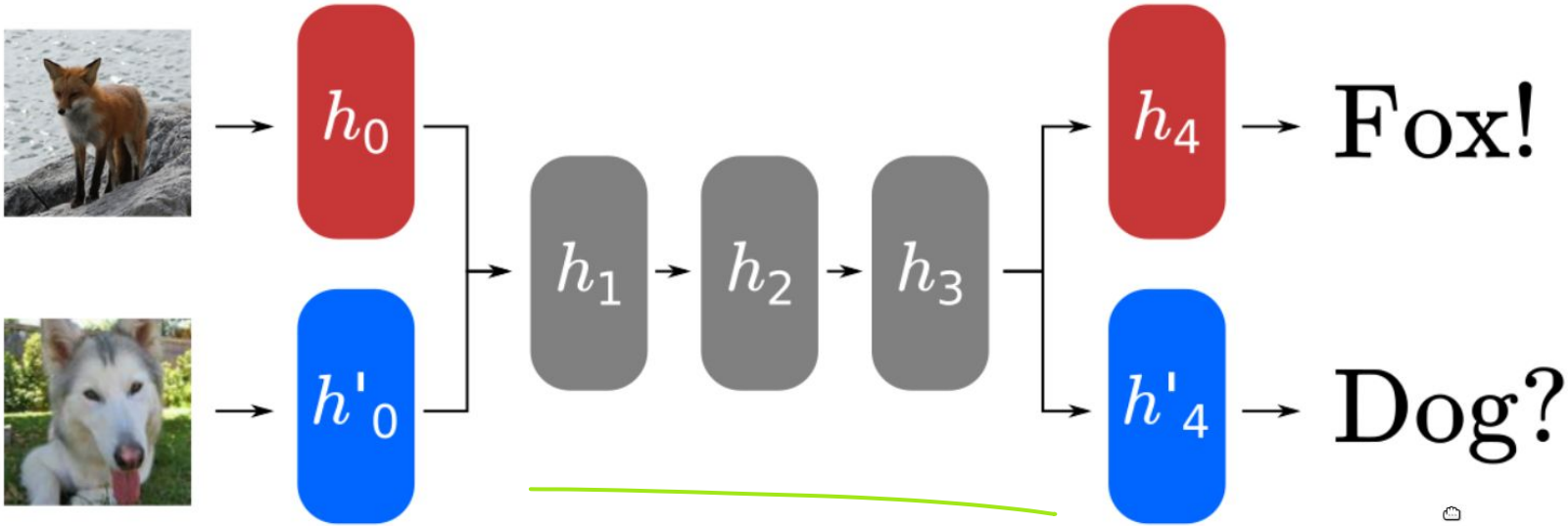


Role of Data augmentation

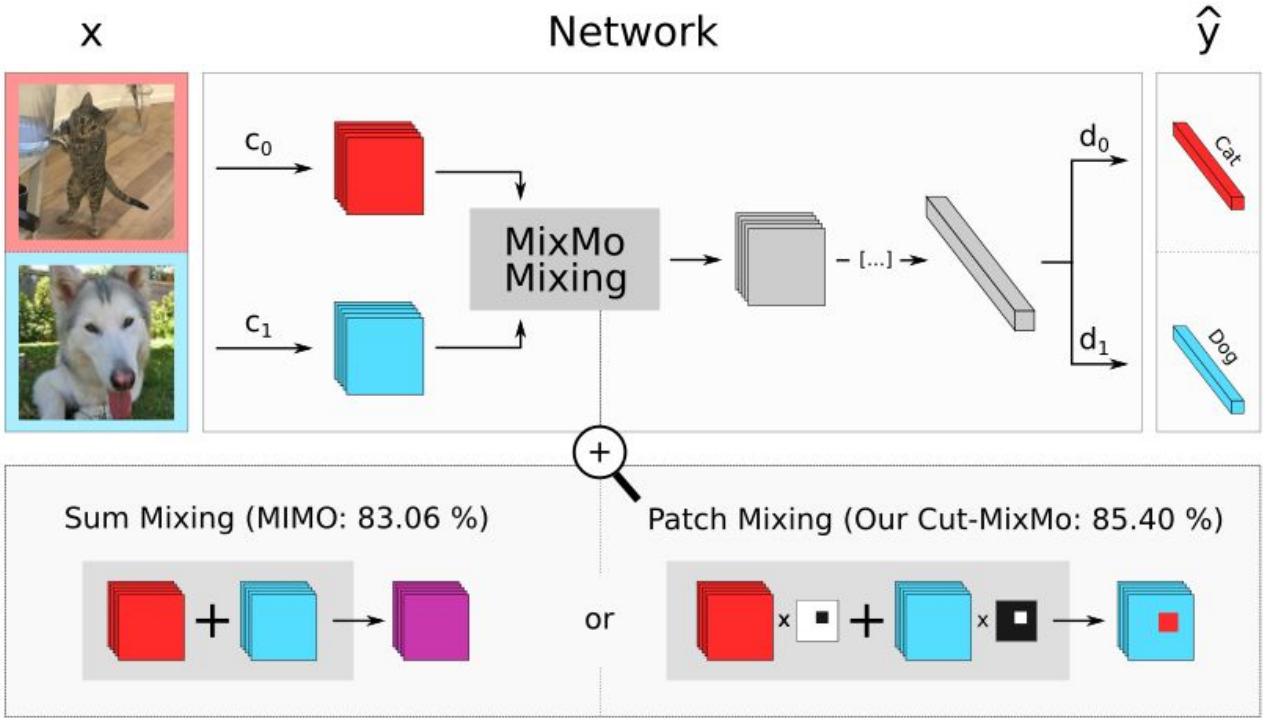


- Role of DA: Smooth out the decision boundaries
  - Reduce out of distribution situations
  - **Does not require realism**

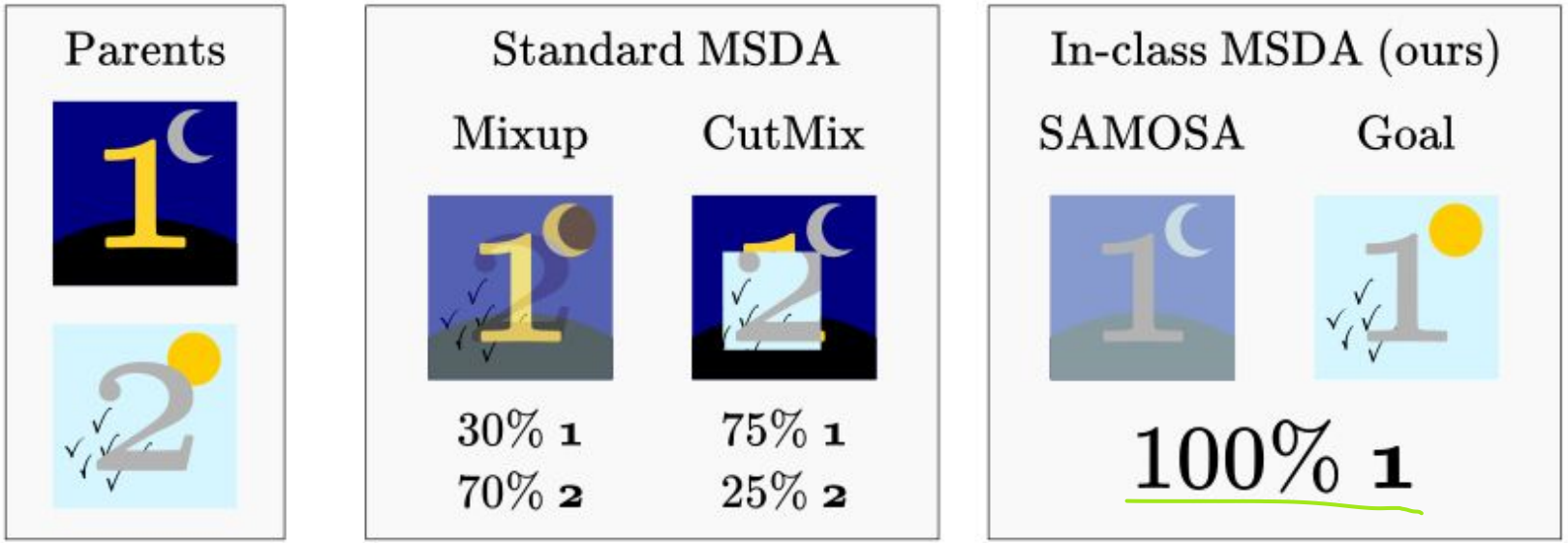
MIMO Mixing Data Augmentation



MIMO Mixing Data Augmentation



In-class MSDA



Takeaway

- Augment the dataset size
  - To avoid overfitting
  - To teach some biases to the model

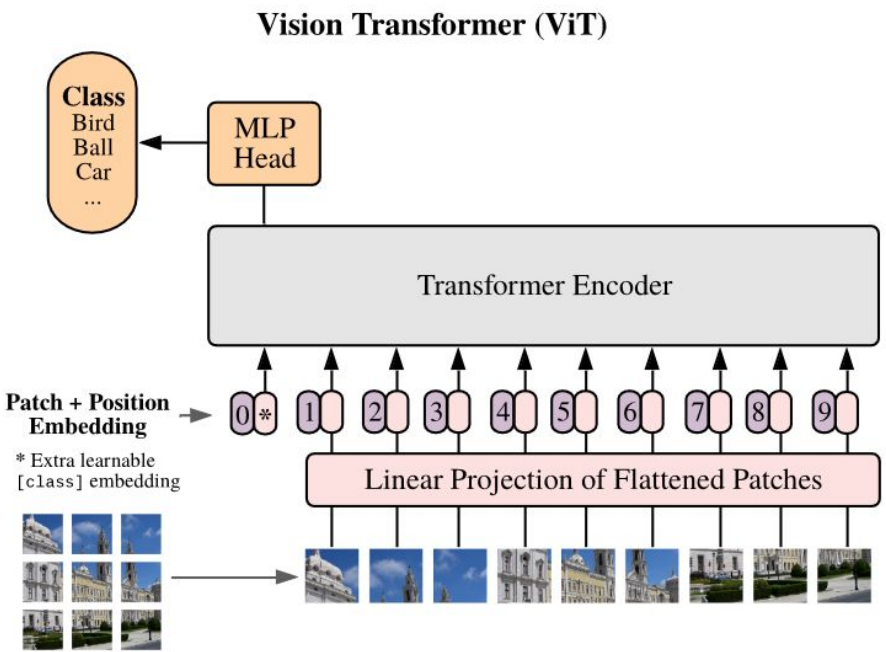
- Classical paradigm
  - Change the image, Keep the label

- Mixing augmentations
  - Combine inputs -> mix the labels

- **Modify image in dataloader transforms or in the loop!**

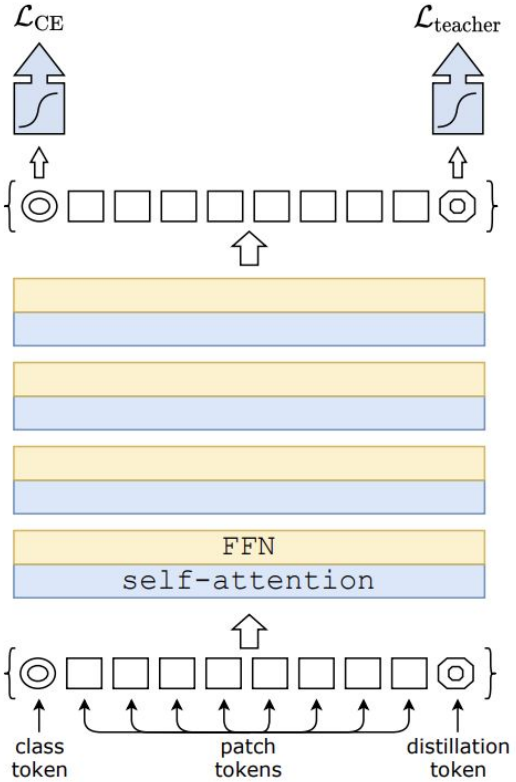
Case study: How ViTs were made to work without huge datasets

Vision Transformer (Dosovitskiy et al, 2021)



- Token = Patch
  - Rich features
- Accumulate relevant info from other tokens
- State of the Art for Computer Vision
  - Data hungry

Data efficient Image T. (Touvron et al. 2021)



- State of the Art for Computer Vision
  - Data hungry
- Solution: Strong regularization
  - Works with “only” ImageNet data

Data efficient Image T. (Touvron et al. 2021)

UNIVERSITÉ CÔTE D'AZUR

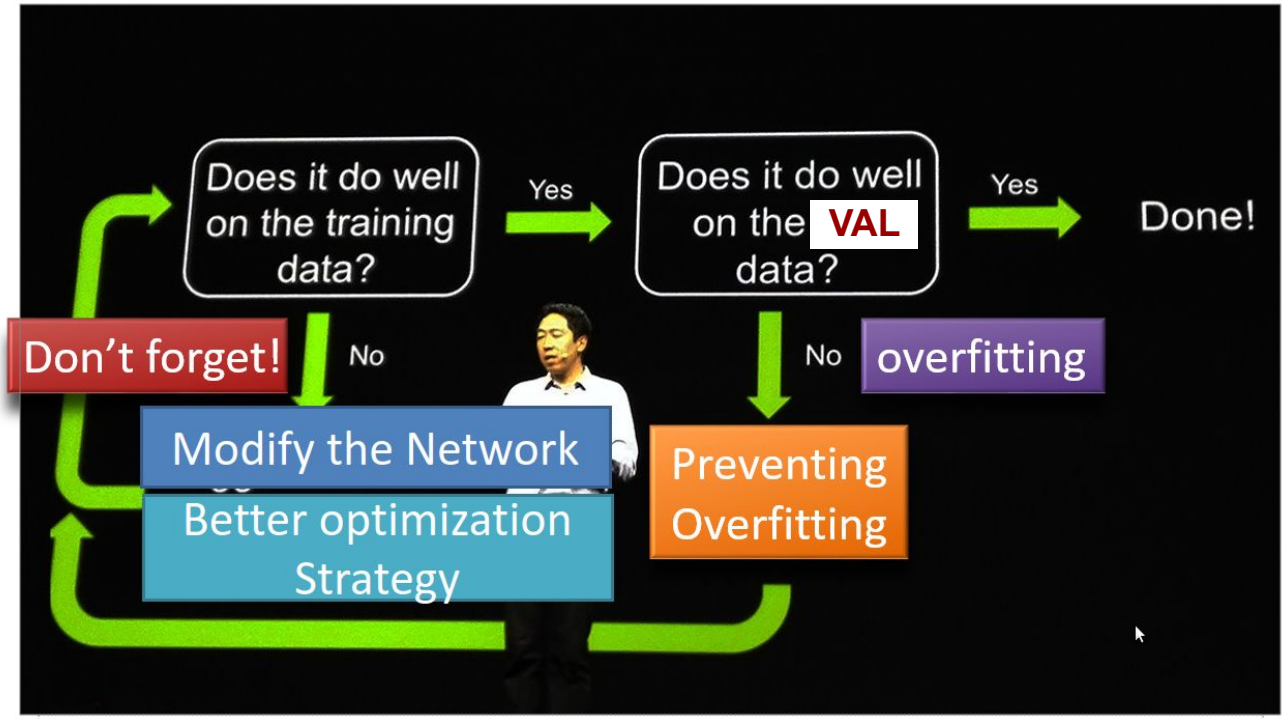
Ablation on ↓	Pre-training	Fine-tuning	Rand-Augment	AutoAug	Mixup	CutMix	Erasing	Stoch. Depth	Repeated Aug.	Dropout	Exp. Moving Avg.	top-1 accuracy: pre-trained 224^2	top-1 accuracy: fine-tuned 384^2
none: DeiT-B	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✗	✗	81.8 ±0.2	83.1 ±0.1
optimizer	SGD	adamw	✓	✗	✓	✓	✓	✓	✓	✗	✗	74.5	77.3
	adamw	SGD	✓	✗	✓	✓	✓	✓	✓	✗	✗	81.8	83.1
data augmentation	adamw	adamw	✗	✗	✓	✓	✓	✓	✓	✗	✗	79.6	80.4
	adamw	adamw	✗	✓	✓	✓	✓	✓	✓	✗	✗	81.2	81.9
	adamw	adamw	✓	✗	✗	✓	✓	✓	✓	✗	✗	78.7	79.8
	adamw	adamw	✓	✗	✓	✗	✓	✓	✓	✗	✗	80.0	80.6
	adamw	adamw	✓	✗	✗	✗	✓	✓	✓	✗	✗	75.8	76.7
regularization	adamw	adamw	✓	✗	✓	✓	✗	✓	✓	✗	✗	4.3*	0.1
	adamw	adamw	✓	✗	✓	✓	✓	✗	✓	✗	✗	3.4*	0.1
	adamw	adamw	✓	✗	✓	✓	✓	✓	✗	✗	✗	76.5	77.4
	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✗	81.3	83.1
	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✗	✓	81.9	83.1

CNNs also benefit (Wightmann et al. 2021)

UNIVERSITÉ CÔTE D'AZUR

	Previous approaches					Ours		
Procedure →	ResNet	PyTorch	FixRes	DeiT	FAMS (×4)	A1	A2	A3
Reference	[13]	[1]	[48]	[45]	[10]			
Train Res	224	224	224	224	224	224	224	160
Test Res	224	224	224	224	224	224	224	224
Epochs	90	90	120	300	400	600	300	100
# of forward pass	450k	450k	300k	375k	500k	375k	188k	63k
Batch size	256	256	512	1024	1024	2048	2048	2048
Optimizer	SGD-M	SGD-M	SGD-M	AdamW	SGD-M	LAMB	LAMB	LAMB
LR	0.1	0.1	0.2	1×10^{-3}	2.0	5×10^{-3}	5×10^{-3}	8×10^{-3}
LR decay	step	step	step	cosine	step	cosine	cosine	cosine
decay rate	0.1	0.1	0.1	-	$0.02^{t/400}$	-	-	-
decay epochs	30	30	30	-	1	-	-	-
Weight decay	10^{-4}	10^{-4}	10^{-4}	0.05	10^{-4}	0.01	0.02	0.02
Warmup epochs	✗	✗	✗	5	5	5	5	5
Label smoothing ε	✗	✗	✗	0.1	0.1	0.1	✗	✗
Dropout	✗	✗	✗	✗	✗	✗	✗	✗
Stoch. Depth	✗	✗	✗	0.1	✗	0.05	0.05	✗
Repeated Aug	✗	✗	✓	✓	✗	✓	✓	✗
Gradient Clip.	✗	✗	✗	✗	✗	✗	✗	✗
H. flip	✓	✓	✓	✓	✓	✓	✓	✓
RRC	✗	✓	✓	✓	✓	✓	✓	✓
Rand Augment	✗	✗	✗	9/0.5	✗	7/0.5	7/0.5	6/0.5
Auto Augment	✗	✗	✗	✗	✓	✗	✗	✗
Mixup alpha	✗	✗	✗	0.8	0.2	0.2	0.1	0.1
Cutmix alpha	✗	✗	✗	1.0	✗	1.0	1.0	1.0
Erasing prob.	✗	✗	✗	0.25	✗	✗	✗	✗
ColorJitter	✗	✓	✓	✗	✗	✗	✗	✗
PCA lighting	✓	✗	✗	✗	✗	✗	✗	✗
SWA	✗	✗	✗	✗	✓	✗	✗	✗
EMA	✗	✗	✗	✗	✗	✗	✗	✗
Test crop ratio	0.875	0.875	0.875	0.875	0.875	0.95	0.95	0.95
CE loss	✓	✓	✓	✓	✓	✗	✗	✗
BCE loss	✗	✗	✗	✗	✗	✓	✓	✓
Mixed precision	✗	✗	✗	✓	✓	✓	✓	✓
Top-1 acc.	75.3%	76.1%	77.0%	78.4%	79.5%	80.4%	79.8%	78.1%

Conclusion


Does it do well on the training data?
Yes
Does it do well on the VAL data?
Yes
Done!
Don't forget!
No
No
overfitting
Modify the Network
Better optimization Strategy
Preventing Overfitting