Introduction to Deep Learning and Transfer Learning

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

2022-02-07

Introduction to Deep Learning and Transfer Learning

Introduction to Deep Learning and Transfer Learning

Global formalism

Input/output

- Goal: infer a function from an input (often tensor) space to an output (often tensor) space, $\mathbf{y} = f(\mathbf{x})$,
- Example: input can be an image, output a vector where the largest value indicate the category the image belongs to.

Error/Loss

- Loss $\mathcal{L}$: nonnegative measure of the discrepancy between expected output $\hat{\mathbf{y}}$ and obtained output $\mathbf{y}$.
- Example: output should be $[0, 1]$ but is $[0.2, 0.8]$.

Parameters

- $f = f_W$ contains **parameters W** to be trained,
- In most cases, an ideal f_W exists but is **hard to find in practice**,
- Learning is a **regression ill-posed** problem.

Global formalism
Input/output
Goal: infer a function from an input (often tensor) space to an output (often tensor) space, $\mathbf{y} = f(\mathbf{x})$,
Example: input can be an image, output a vector where the largest value indicate the category the image belongs to.
Error/Loss
Loss $\mathcal{L}$: nonnegative measure of the discrepancy between expected output $\hat{\mathbf{y}}$ and obtained output $\mathbf{y}$.
Example: output should be $[0, 1]$ but is $[0.2, 0.8]$.
Parameters
$f = f_W$ contains parameters W to be trained,
In most cases, an ideal f_W exists but is hard to find in practice,
Learning is a regression ill-posed problem.

Loss: it's a way to tell the model when it is wrong and train the model accordingly. The model contains parameters (model weights and bias) and usually, given a task, an optimal set of parameters exist but again finding it is ill posed problem (many solutions exists

Global formalism

Input/output

- Goal: infer a function from an input (often tensor) space to an output (often tensor) space, $\mathbf{y} = f(\mathbf{x})$,
- Example: input can be an image, output a vector where the largest value indicate the category the image belongs to.

Error/Loss

- Loss $\mathcal{L}$: nonnegative measure of the discrepancy between expected output $\hat{\mathbf{y}}$ and obtained output $\mathbf{y}$.
- Example: output should be $[0, 1]$ but is $[0.2, 0.8]$.

Parameters

- $f = f_W$ contains **parameters W** to be trained,
- In most cases, an ideal f_W exists but is **hard to find in practice**,
- Learning is a **regression ill-posed** problem.

Global formalism
Input/output
Goal: infer a function from an input (often tensor) space to an output (often tensor) space, $\mathbf{y} = f(\mathbf{x})$,
Example: input can be an image, output a vector where the largest value indicate the category the image belongs to.
Error/Loss
Loss $\mathcal{L}$: nonnegative measure of the discrepancy between expected output $\hat{\mathbf{y}}$ and obtained output $\mathbf{y}$.
Example: output should be $[0, 1]$ but is $[0.2, 0.8]$.
Parameters
$f = f_W$ contains parameters W to be trained,
In most cases, an ideal f_W exists but is hard to find in practice,
Learning is a regression ill-posed problem.

Loss: it's a way to tell the model when it is wrong and train the model accordingly. The model contains parameters (model weights and bias) and usually, given a task, an optimal set of parameters exist but again finding it is ill posed problem (many solutions exists

Global formalism

Input/output

- Goal: infer a function from an input (often tensor) space to an output (often tensor) space, $\mathbf{y} = f(\mathbf{x})$,
- Example: input can be an image, output a vector where the largest value indicate the category the image belongs to.

Error/Loss

- Loss $\mathcal{L}$: nonnegative measure of the discrepancy between expected output $\hat{\mathbf{y}}$ and obtained output $\mathbf{y}$.
- Example: output should be $[0, 1]$ but is $[0.2, 0.8]$.

Parameters

- $f = f_W$ contains **parameters W** to be trained,
- In most cases, an ideal f_W exists but is **hard to find in practice**,
- Learning is a **regression ill-posed** problem.

Global formalism
Input/output
Goal: infer a function from an input (often tensor) space to an output (often tensor) space, $\mathbf{y} = f(\mathbf{x})$,
Example: input can be an image, output a vector where the largest value indicate the category the image belongs to.
Error/Loss
Loss $\mathcal{L}$: nonnegative measure of the discrepancy between expected output $\hat{\mathbf{y}}$ and obtained output $\mathbf{y}$.
Example: output should be $[0, 1]$ but is $[0.2, 0.8]$.
Parameters
$f = f_W$ contains **parameters W** to be trained,
In most cases, an ideal f_W exists but is **hard to find in practice**,
Learning is a **regression ill-posed** problem.

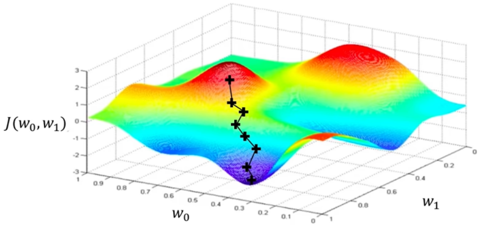
Loss: it's a way to tell the model when it is wrong and train the model accordingly. The model contains parameters (model weights and bias) and usually, given a task, an optimal set of parameters exist but again finding it is ill posed problem (many solutions exists

Global formalism

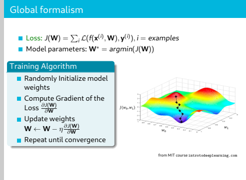
- Loss: $J(\mathbf{W}) = \sum_i \mathcal{L}(f(\mathbf{x}^{(i)}, \mathbf{W}), \mathbf{y}^{(i)}), i = examples$
- Model parameters: $\mathbf{W}^* = argmin(J(\mathbf{W}))$

Training Algorithm

- Randomly Initialize model weights
- Compute Gradient of the Loss $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
- Update weights $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
- Repeat until convergence



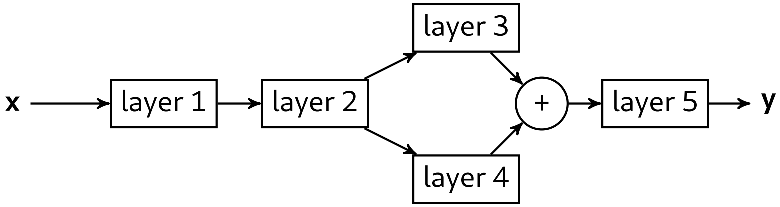
from MIT course `introtodeeplearning.com`

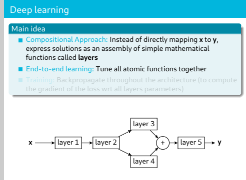


The total loss J (Empirical Risk, Objective function) is the average of Loss for each input/example and the optimal model parameters are those that minimize it. But how to find them? In other words, how to train the model? Here is a simplified description of the training algorithm at the base of modern DL, gradient descent. Repeat until reaching a local minimum (as illustrated in the figure for a simple example where we have only 2 parameters. We'll see that the function becomes much more complicated for millions of parameters -modern neural networks.)

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



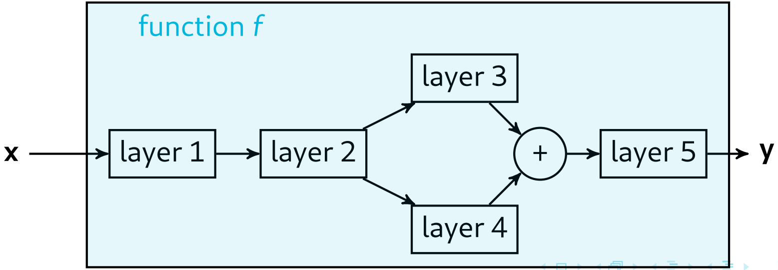


DL at is core is the ability to learn higher and higher level representations or features from data in a end to end fashion. How? By means of a compositional approach of simple mathematical functions (layers). Representation are useful to interpret data: ideally the final representation should be easy to deal with (to classify, to generate data from...). What is new about DL is that we do that in a end-to-end fashion starting from raw data. Also, in DL, we use deep architectures with hidden layers to approximate any complex function f. So the fundamental blocks of NN are layers, each layers has is own parameters (weights and bias) that need to be trained.

Deep learning

Main idea

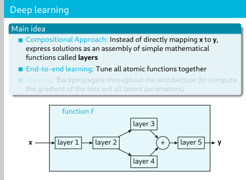
- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



2022-02-07

Introduction to Deep Learning and Transfer Learning

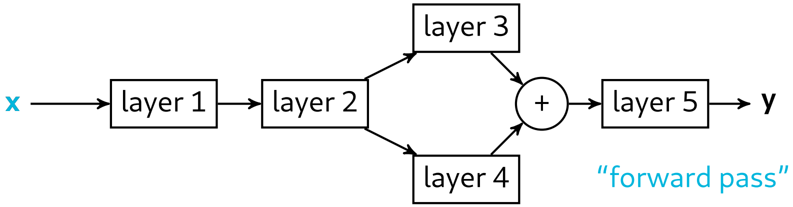
└─Deep learning

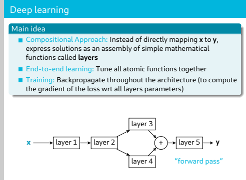


Deep learning

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



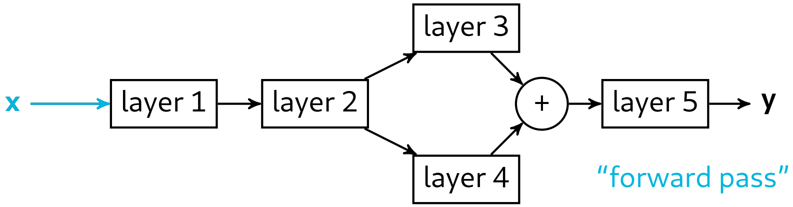


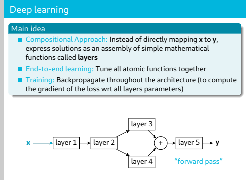
As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Deep learning

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

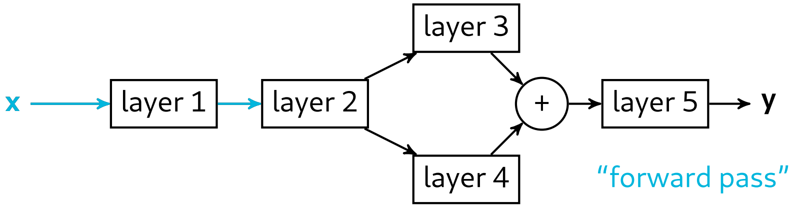


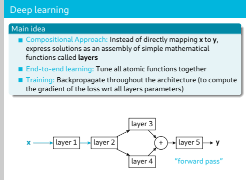


As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



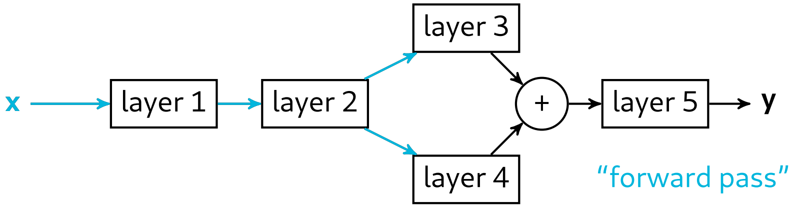


As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Deep learning

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



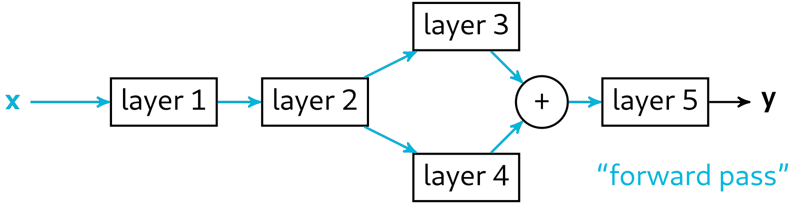
Deep learning
Main idea
Compositional Approach: Instead of directly mapping x to y, express solutions as an assembly of simple mathematical functions called layers
End-to-end learning: Tune all atomic functions together
Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)
x layer 1 layer 2 layer 3 layer 4 + layer 5 y "forward pass"

As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Deep learning

Main idea

- Compositional Approach: Instead of directly mapping $\mathbf{x}$ to $\mathbf{y}$, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



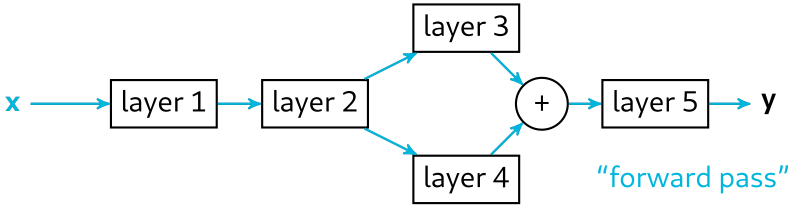
Deep learning
Main idea
Compositional Approach: Instead of directly mapping x to y, express solutions as an assembly of simple mathematical functions called layers
End-to-end learning: Tune all atomic functions together
Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)
x layer 1 layer 2 layer 3 layer 4 + layer 5 y "forward pass"

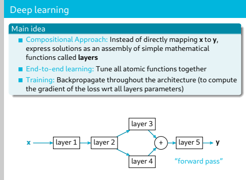
As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Deep learning

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



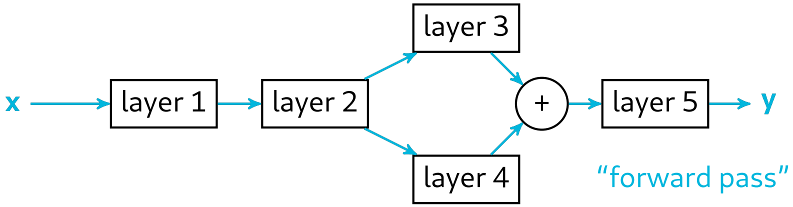


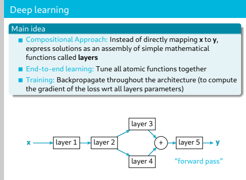
As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Deep learning

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



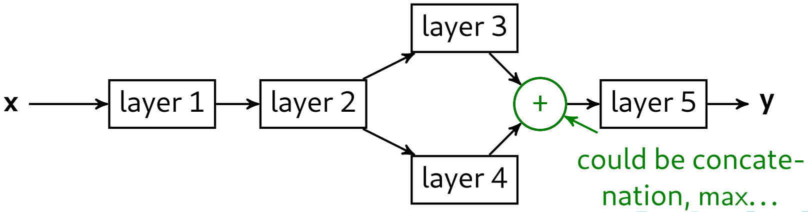


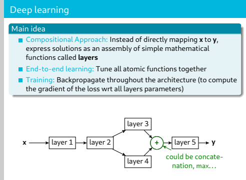
As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Deep learning

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

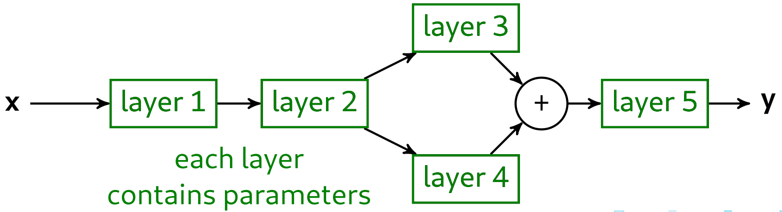


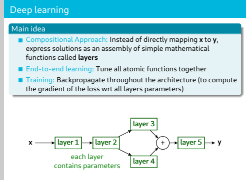


As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

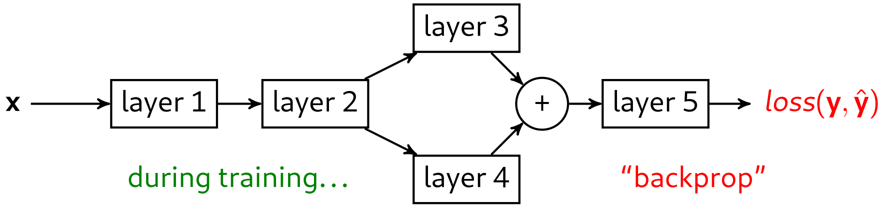


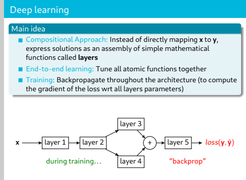


As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Main idea

- Compositional Approach: Instead of directly mapping $\mathbf{x}$ to $\mathbf{y}$, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

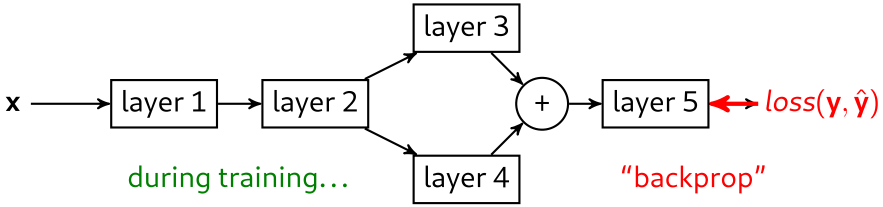


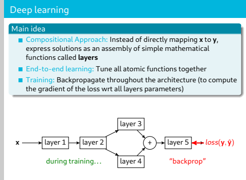


As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Main idea

- Compositional Approach: Instead of directly mapping $\mathbf{x}$ to $\mathbf{y}$, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

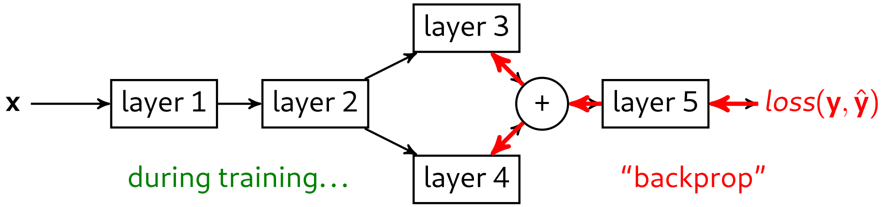


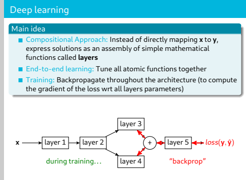


As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Main idea

- Compositional Approach: Instead of directly mapping $\mathbf{x}$ to $\mathbf{y}$, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

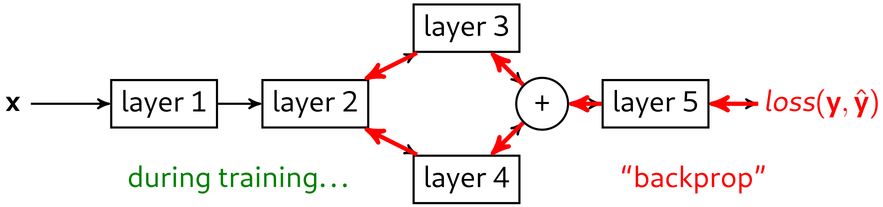


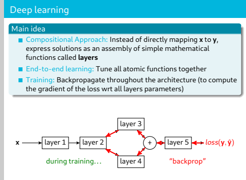


As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Main idea

- Compositional Approach: Instead of directly mapping $\mathbf{x}$ to $\mathbf{y}$, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

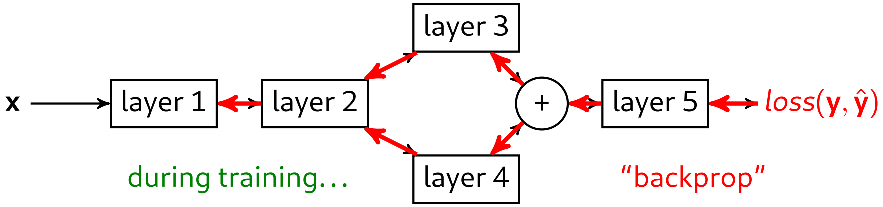


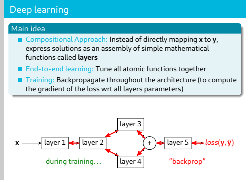


As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)

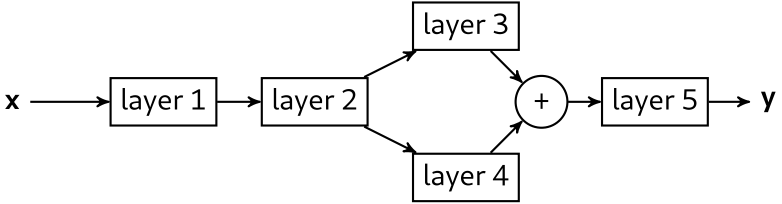




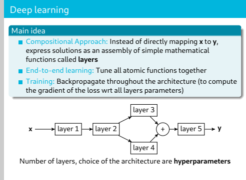
As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Main idea

- Compositional Approach: Instead of directly mapping **x** to **y**, express solutions as an assembly of simple mathematical functions called **layers**
- End-to-end learning: Tune all atomic functions together
- Training: Backpropagate throughout the architecture (to compute the gradient of the loss wrt all layers parameters)



Number of layers, choice of the architecture are **hyperparameters**



As we have seen the model is trained by calculating the gradient of the loss wrt to each layer parameters. How do we calculate gradients? Using backpropagation: efficient way to compute the gradient of the loss with respect to different layers parameters, using the derivative chain rule. The model weights are then updated with the rule we have seen $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ so that those responsible for producing the right output are increased, and the other decreased.

Some additional details

Layers

- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:
    - Can be agnostic of the structure: fully-connected layers,
    - Can be structure-dependent: convolutional layers.

Non linearity: approximate complex functions! Otherwise combination of linear transformations.

Some additional details

Layers

- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:
    - Can be agnostic of the structure: **fully-connected layers**,
    - Can be structure-dependent: **convolutional layers**.

2022-02-07

Introduction to Deep Learning and Transfer Learning

└─Some additional details

Some additional details

Layers

- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:

Layers

- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:
    - Can be agnostic of the structure: **fully-connected layers**,
    - Can be structure-dependent: **convolutional layers**.

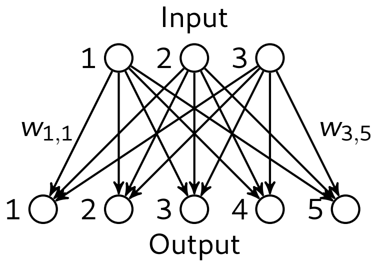
Some additional details
Layers
x ↦ h(Wx + b).
h is a nonlinear parameterwise function (often without parameters),
W is a tensor:
Can be agnostic of the structure: fully-connected layers,
Can be structure-dependent: convolutional layers.

Some additional details

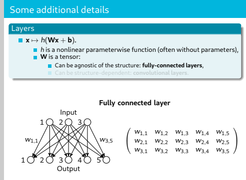
Layers

- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:
    - Can be agnostic of the structure: **fully-connected layers**,
    - Can be structure-dependent: **convolutional layers**.

Fully connected layer



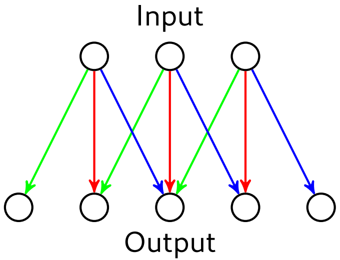
$$\begin{pmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{2,5} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & w_{3,5} \end{pmatrix}$$



Layers

- $\mathbf{x} \mapsto h(\mathbf{W}\mathbf{x} + \mathbf{b})$.
  - h is a nonlinear parameterwise function (often without parameters),
  - $\mathbf{W}$ is a tensor:
    - Can be agnostic of the structure: **fully-connected layers**,
    - Can be structure-dependent: **convolutional layers**.

Convolutional layer

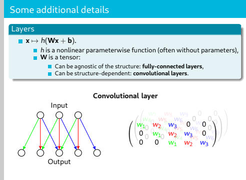


$$\begin{pmatrix} w_1 & w_2 & w_3 & 0 & 0 \\ 0 & w_1 & w_2 & w_3 & 0 \\ 0 & 0 & w_1 & w_2 & w_3 \end{pmatrix}$$

2022-02-07

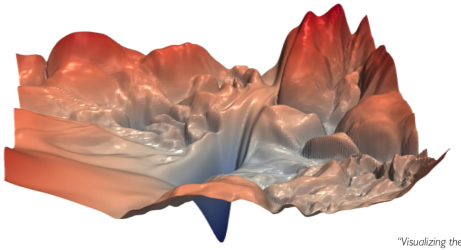
Introduction to Deep Learning and Transfer Learning

└Some additional details



Some additional details

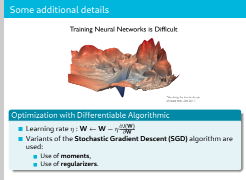
Training Neural Networks is Difficult



"Visualizing the loss landscape of neural nets". Dec 2017.

Optimization with Differentiable Algorithmic

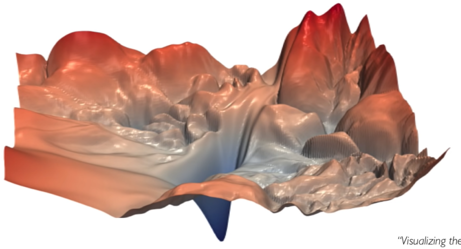
- Learning rate η : $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
- Variants of the **Stochastic Gradient Descent (SGD)** algorithm are used:
  - Use of **moments**,
  - Use of **regularizers**.



Training a NN is a challenging task: this picture represents the loss landscape of a typical DL network with million of parameters, something very different from the version we have seen before for 2 parameters, extremely complex and with many local minima. Optimization depends of different factors but one of the most crucial one is the learning rate (the fraction of the gradient that is subtracted from the loss) as it determines the convergence of the SGD: it should be large enough to avoid local minima, but small enough to converge. Most of modern implementation use an adaptive lr (increase, decrease during training): try out different adaptive schemes during the lab! Also different optimizers, all variants of SGD can be explored. To increase generalization (or in other words, avoid overfitting) different regularizations techniques. Momentum: help reduce variance: accumulates a decaying moving average of past gradients so the gradient step depends on how aligned past gradients are.

Some additional details

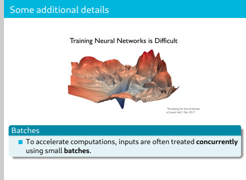
Training Neural Networks is Difficult



“Visualizing the loss landscape of neural nets”. Dec 2017.

Batches

- To accelerate computations, inputs are often treated **concurrently** using small **batches**.



Backprop is computationally intensive if performed for each data example. One way to accelerate computation is the compute the gradient of batches (or small group) of training examples. This also gives a better estimate of the gradient, allows for paralellization and higher lr. Of course there is a tradeoff between higher speed (large batches) and better generalization: batch size is a hyperparameter itself. Recap: batch: gradient step, epoch: iteration over the entire dataset (ensemble of batches)

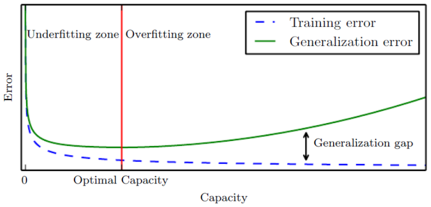
Generalization vs Overfitting

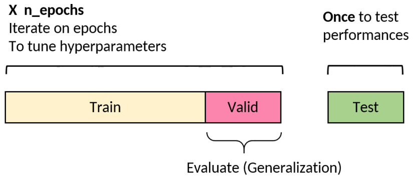
Learning Objectives

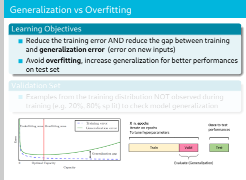
- Reduce the training error AND reduce the gap between training and **generalization error** (error on new inputs)
- Avoid **overfitting**, increase generalization for better performances on test set

Validation Set

- Examples from the training distribution NOT observed during training (e.g. 20%, 80% sp lit) to check model generalization







All these hyperparameters choices and regularization techniques have the objective to increase generalization or reduce overfitting. The typical model learning curves are showed in the left figure. In the left area when both train and generalization error are high we are in an underfitting regime: the model is not able to express the complexity of the dataset. When the gap between the generalization error and the train error increases we are specializing to much on the dataset (Overfitting regime). One way to assess this is to evaluate the performance on a validation set (split as figure on the right) and one popular regularization technique is the early stopping: stop training at the inflection point. The are many other regularization techniques (dropout: randomy set some model units to zero, also increases robustness; normalization of inputs, batch norm: normalization for intermediate features deeper in the network.)

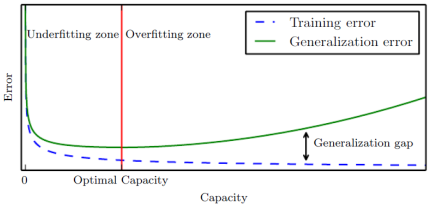
Generalization vs Overfitting

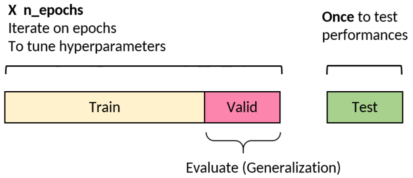
Learning Objectives

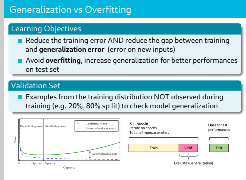
- Reduce the training error AND reduce the gap between training and **generalization error** (error on new inputs)
- Avoid **overfitting**, increase generalization for better performances on test set

Validation Set

- Examples from the training distribution NOT observed during training (e.g. 20%, 80% sp lit) to check model generalization



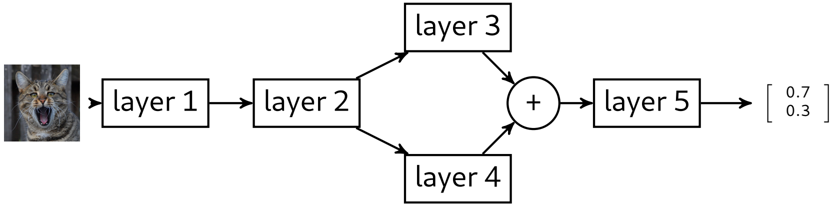


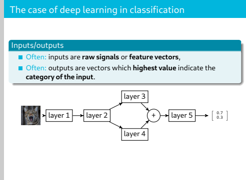


All these hyperparameters choices and regularization techniques have the objective to increase generalization or reduce overfitting. The typical model learning curves are showed in the left figure. In the left area when both train and generalization error are high we are in an underfitting regime: the model is not able to express the complexity of the dataset. When the gap between the generalization error and the train error increases we are specializing to much on the dataset (Overfitting regime). One way to assess this is to evaluate the performance on a validation set (split as figure on the right) and one popular regularization technique is the early stopping: stop training at the inflection point. The are many other regularization techniques (dropout: randomy set some model units to zero, also increases robustness; normalization of inputs, batch norm: normalization for intermediate features deeper in the network.)

Inputs/outputs

- Often: inputs are **raw signals** or **feature vectors**,
- Often: outputs are vectors which **highest value** indicate the **category of the input**.

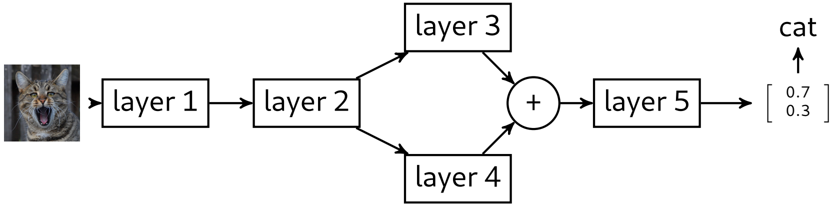


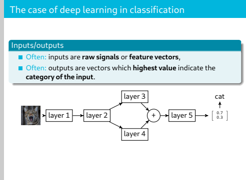


Softmax to generate a probability from numbers. Cross Entropy Loss (Shannon 50 years ago) compare how different the predicted and labels distribution are.

Inputs/outputs

- Often: inputs are **raw signals** or **feature vectors**,
- Often: outputs are vectors which **highest value** indicate the **category of the input**.



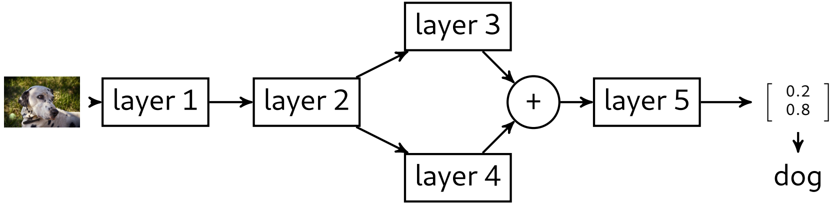


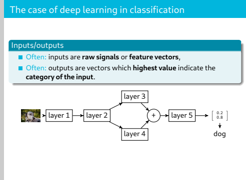
Softmax to generate a probability from numbers. Cross Entropy Loss (Shannon 50 years ago) compare how different the predicted and labels distribution are.

The case of deep learning in classification

Inputs/outputs

- Often: inputs are **raw signals** or **feature vectors**,
- Often: outputs are vectors which **highest value** indicate the **category of the input**.



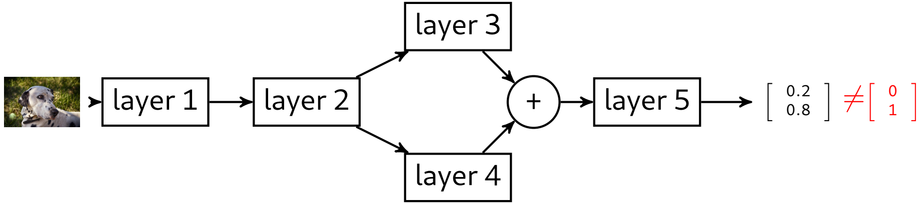


Softmax to generate a probability from numbers. Cross Entropy Loss (Shannon 50 years ago) compare how different the predicted and labels distribution are.

The case of deep learning in classification

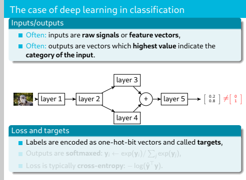
Inputs/outputs

- Often: inputs are **raw signals** or **feature vectors**,
- Often: outputs are vectors which **highest value** indicate the **category of the input**.



Loss and targets

- Labels are encoded as one-hot-bit vectors and called **targets**,
- Outputs are **softmaxed:** $\mathbf{y}_i \leftarrow \exp(\mathbf{y}_i)/\sum_j \exp(\mathbf{y}_j)$,
- Loss is typically **cross-entropy:** $-\log(\hat{\mathbf{y}}^\top \mathbf{y})$.

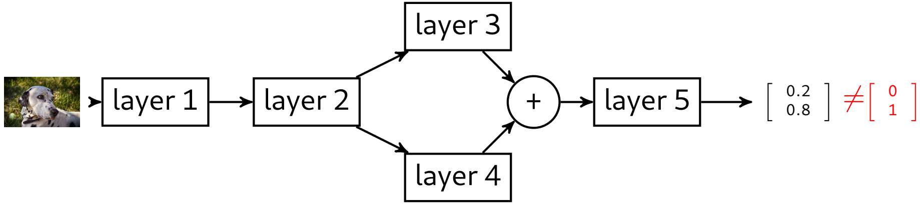


Softmax to generate a probability from numbers. Cross Entropy Loss (Shannon 50 years ago) compare how different the predicted and labels distribution are.

The case of deep learning in classification

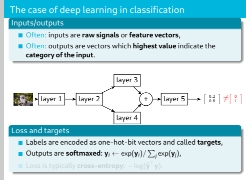
Inputs/outputs

- Often: inputs are **raw signals** or **feature vectors**,
- Often: outputs are vectors which **highest value** indicate the **category of the input**.



Loss and targets

- Labels are encoded as one-hot-bit vectors and called **targets**,
- Outputs are **softmaxed:** $\mathbf{y}_i \leftarrow \exp(\mathbf{y}_i) / \sum_j \exp(\mathbf{y}_j)$,
- Loss is typically **cross-entropy**: $-\log(\hat{\mathbf{y}}^\top \mathbf{y})$.

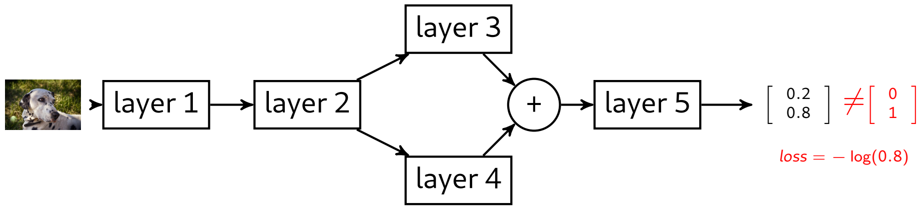


Softmax to generate a probability from numbers. Cross Entropy Loss (Shannon 50 years ago) compare how different the predicted and labels distribution are.

The case of deep learning in classification

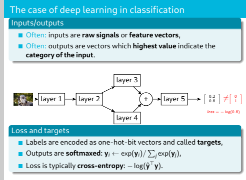
Inputs/outputs

- Often: inputs are **raw signals** or **feature vectors**,
- Often: outputs are vectors which **highest value** indicate the **category of the input**.



Loss and targets

- Labels are encoded as one-hot-bit vectors and called **targets**,
- Outputs are **softmaxed**: $\mathbf{y}_i \leftarrow \exp(\mathbf{y}_i) / \sum_j \exp(\mathbf{y}_j)$,
- Loss is typically **cross-entropy**: $-\log(\hat{\mathbf{y}}^\top \mathbf{y})$.



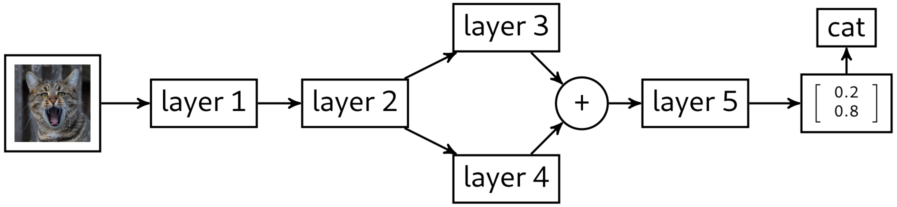
Softmax to generate a probability from numbers. Cross Entropy Loss (Shannon 50 years ago) compare how different the predicted and labels distribution are.

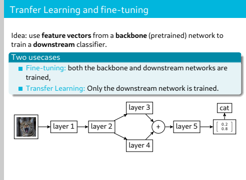
Tranfer Learning and fine-tuning

Idea: use **feature vectors** from a **backbone** (pretrained) network to train a **downstream** classifier.

Two usecases

- Fine-tuning: both the backbone and downstream networks are trained,
- Transfer Learning: Only the downstream network is trained.





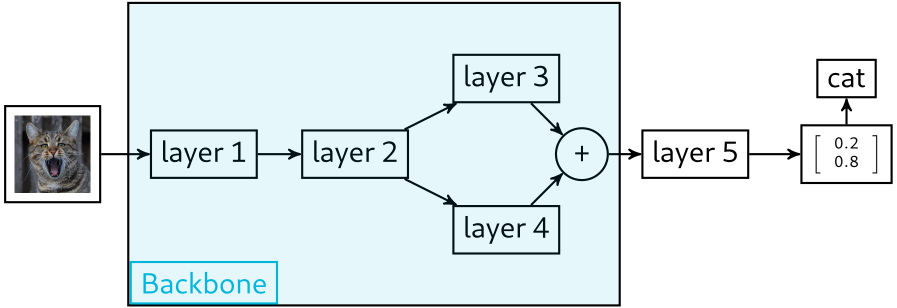
An idea that is extensively applied in computer vision and more generally in DL is transfer learning. Consist in exploiting the knowledge of a network pretrained on a large dataset to adapt it for a novel, usually smaller and more specialized dataset/ classification task on dataset. 2 ways: Fine tuning: retrain for a few epochs the whole network on the novel dataset.

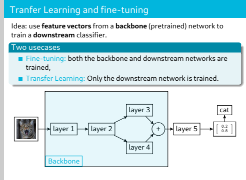
Tranfer Learning and fine-tuning

Idea: use **feature vectors** from a **backbone** (pretrained) network to train a **downstream** classifier.

Two usecases

- Fine-tuning: both the backbone and downstream networks are trained,
- Transfer Learning: Only the downstream network is trained.



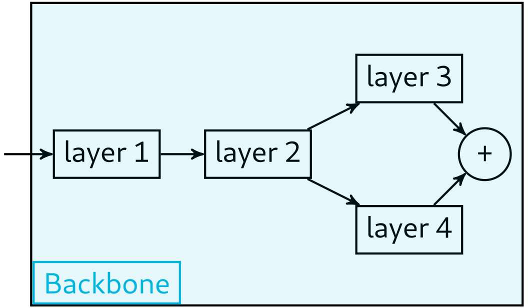


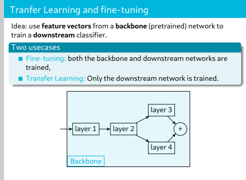
Tranfer Learning and fine-tuning

Idea: use **feature vectors** from a **backbone** (pretrained) network to train a **downstream** classifier.

Two usecases

- Fine-tuning: both the backbone and downstream networks are trained,
- Transfer Learning: Only the downstream network is trained.





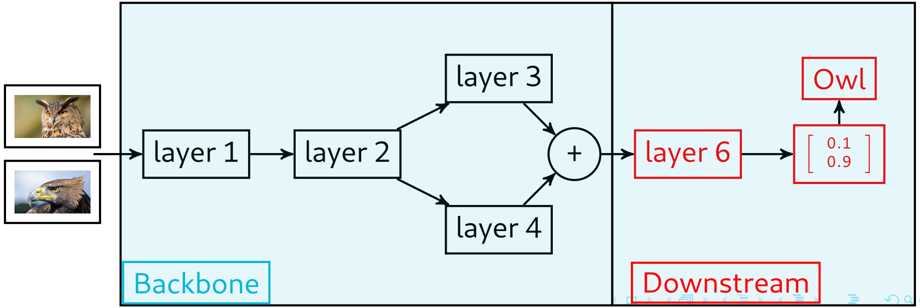
Transfer Learning: chop out from the pretrained network the last dense/fully connected layers: this part is frozen and called backbone and only the final layer(s) are retrained on the novel dataset.

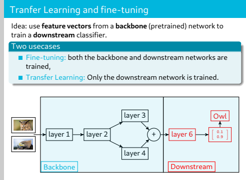
Tranfer Learning and fine-tuning

Idea: use **feature vectors** from a **backbone** (pretrained) network to train a **downstream** classifier.

Two usecases

- Fine-tuning: both the backbone and downstream networks are trained,
- Transfer Learning: Only the downstream network is trained.





These two techniques can obviously be combined! And are very useful to specialise the network for a precise task like for instance classifing birds, using a backbone trained on a big CV dataset such as ImageNet.

Hyperparameters

Architecture

- Number of layers
- Architecture choice (e.g. ResNet, DenseNet, VGG, . . .)

Training

- Learning rate and scheduling
- Regularization (e.g. weight decay)
- Choice of optimizer (e.g. SGD)

2022-02-07

Introduction to Deep Learning and Transfer Learning

└─Hyperparameters

Hyperparameters
Architecture
Number of layers
Architecture choice (e.g. ResNet, DenseNet, VGG, . . .)
Training
Learning rate and scheduling
Regularization (e.g. weight decay)
Choice of optimizer (e.g. SGD)

Hyperparameters

Architecture

- Number of layers
- Architecture choice (e.g. ResNet, DenseNet, VGG, ...)

Training

- Learning rate and scheduling
- Regularization (e.g. weight decay)
- Choice of optimizer (e.g. SGD)

Hyperparameters
Architecture
- Number of layers
- Architecture choice (e.g. ResNet, DenseNet, VGG, ...)
Training
- Learning rate and scheduling
- Regularization (e.g. weight decay)
- Choice of optimizer (e.g. SGD)

Introduction to Deep Learning

- Introduction to Deep Learning in Pytorch
- Train a full DL model from scratch
- Train a downstream model using transfer learning

Project 1 (oral presentation)

Explore one of the following architectures : ResNet, DenseNet, PreActResNet, VGG.
You have to prepare a 10 minutes (+5 min Q&A) presentation for session 2, in which you explain :

- Description of the architecture
- Hyperparameter search and results
- Study the compromise between architecture size, performance and training time.

Lab Session 1 and assignment
Introduction to Deep Learning
- Introduction to Deep Learning in Pytorch
- Train a full DL model from scratch
- Train a downstream model using transfer learning
Project 1 (oral presentation)
Explore one of the following architectures : ResNet, DenseNet, PreActResNet, VGG.
You have to prepare a 10 minutes (+5 min Q&A) presentation for session 2, in which you explain :
- Description of the architecture
- Hyperparameter search and results
- Study the compromise between architecture size, performance and training time.