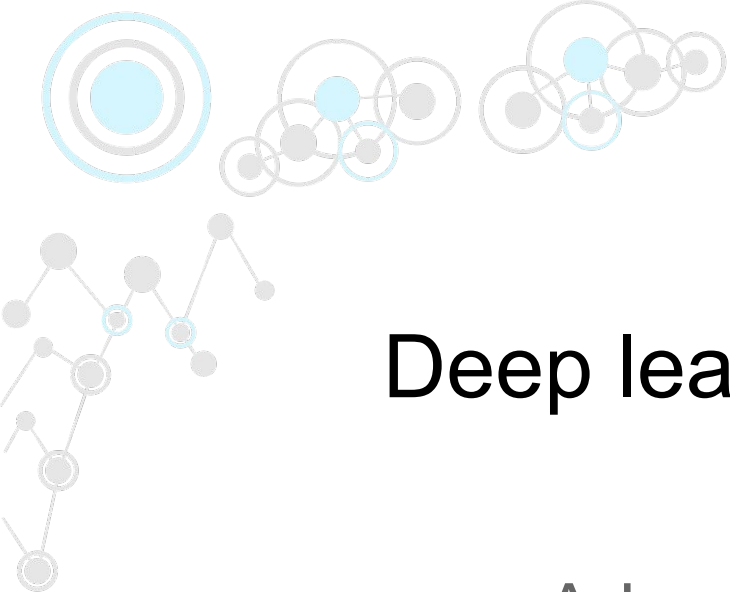


Lecture 4: Deep learning in practice with pytorch

Advanced deep learning

Rémy Sun
remy.sun@inria.fr



About the next few lectures

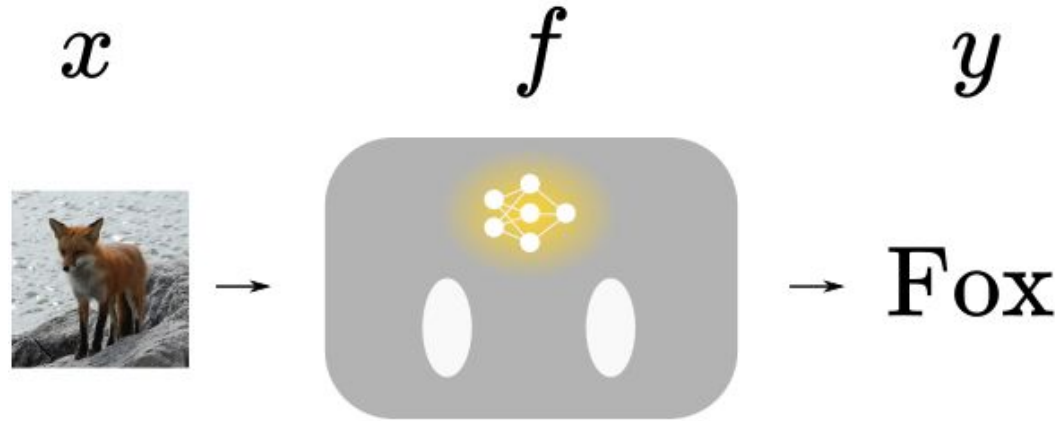
- Goal: In-depth understanding of important Deep Learning staples
 - Reinforce what you have already seen
 - Introduce state of the art models
- This is a hands-on course in pytorch
 - Minimal math
 - Enough to understand
 - Quite a bit of coding
 - Get comfortable with the standard pipeline

- Hand in one or two lab notebooks
 - Questions + (clean) code
 - 1st notebook: Lab 5 on transformers (Next Thursday)
 - To hand in after vacation
- Written exam at end of semester
 - Little to no code
 - A few exercises on toy examples
 - Questions on aspects of deep learning

- L1-2: Overview of Deep Learning (F. Precioso)
- L3-4: Fundamentals of Deep Learning (R. Sun)
- L5-6: Transformers (R. Sun)
- L7: Large models (LLMs, VLMs, Generators) (R. Sun)
- L8: Tricks of the trade (R. Sun)
- L9: Ethics of AI (F. Precioso)
- L10: Intro to generative models (P-A. Mattei)

- Goal: Understand basic deep architectures in-depth
 - Building blocks for everything else in deep learning
 - Deep learning relies on the combination of a lot of very simple blocks
- A few things to take away after these 3 lectures
 - What do we optimize for? How? Why?
 - How do we build and train neural layers?
 - How do they behave?

Refresher on last week

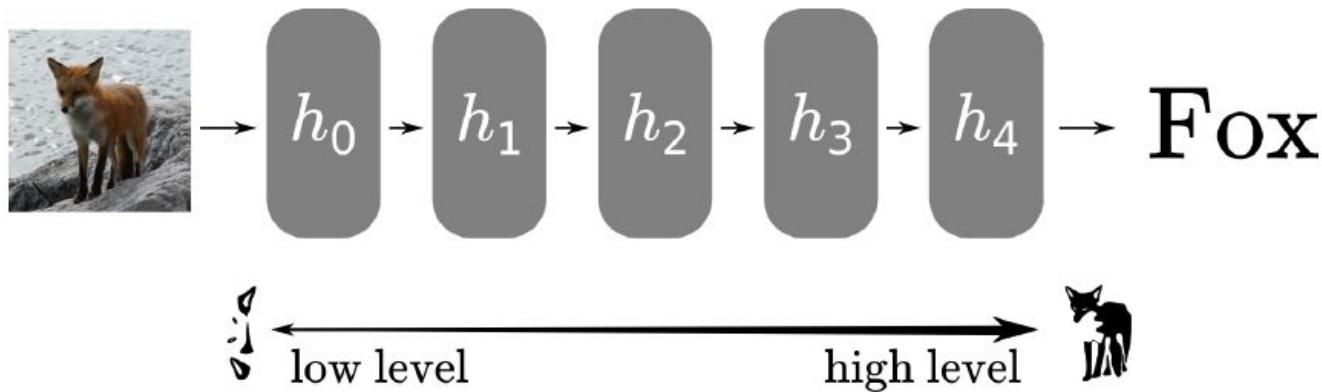


- Find (robot) f that classifies images well
 - Often based on neural networks

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

- Problem: we do not know \mathcal{D} !
 - Solved problem otherwise...
 - Evaluating the risk requires this distribution
- Solution: Use a dataset D of (x,y) sampled from \mathcal{D}
 - **Empirical Risk Minimization**
 - If the (x,y) are i.i.d drawn from \mathcal{D} can be expressed as a mean over the dataset

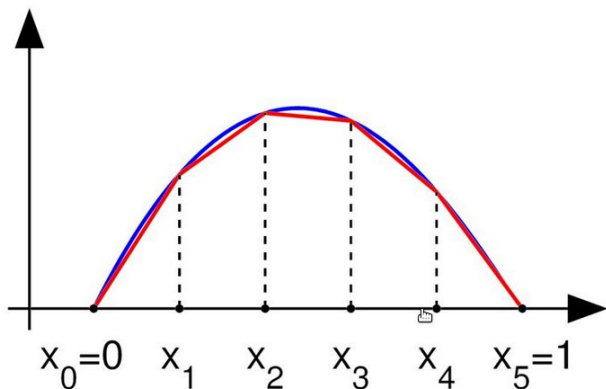
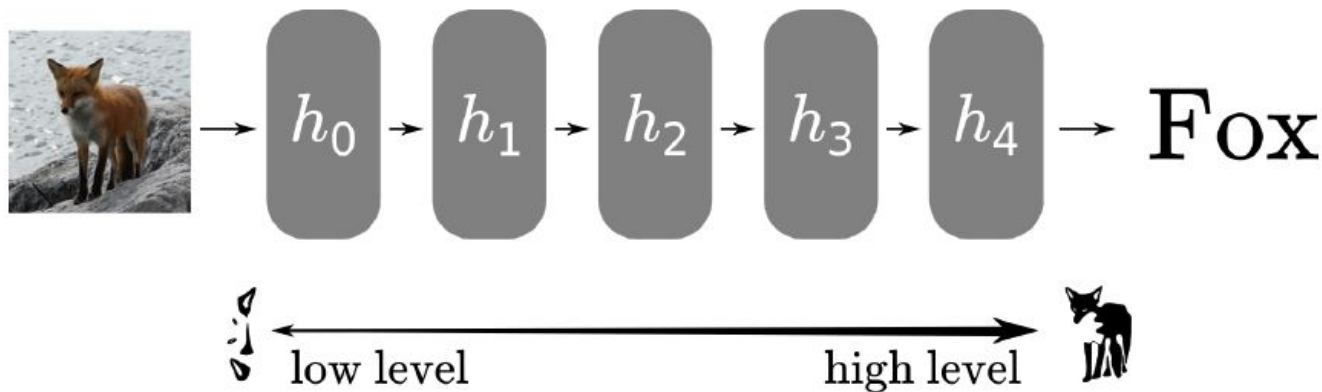
$$\min_{\theta} \hat{\mathcal{R}}_{\theta} = \frac{1}{N} \sum_{i=0, \dots, N-1} l(f_{\theta}(x_i), y_i)$$



- Neural networks are sequences of simple functions

$$f_{\theta} = h_{\theta}^0 \circ h_{\theta}^1 \circ \dots \circ h_{\theta}^{L-1}$$

Let's find the best network then!



$$\min_{\theta} \hat{\mathcal{R}}_{\theta} = \frac{1}{N} \sum_{i=0, \dots, N-1} l(f_{\theta}(x_i), y_i)$$

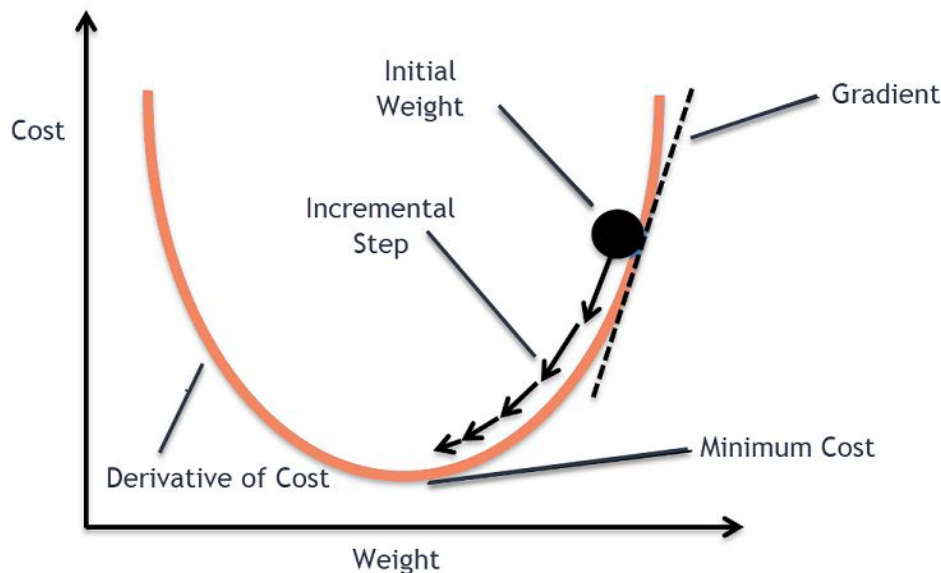
No closed form!

- Iteratively make steps of size η to minimize risk

$$\theta^{t+1} := \theta^t - \eta \nabla_{\theta} \hat{\mathcal{R}}_{\theta}$$

- Elementwise form:

$$\theta_i^{t+1} := \theta_i^t - \eta \frac{\partial \hat{\mathcal{R}}_{\theta}}{\partial \theta_i}$$



$$\theta^{t+1} := \theta^t - \eta \nabla_{\theta} \mathcal{R}_{\theta}(\hat{B})$$

- Requires finding the risk gradient wrt parameters

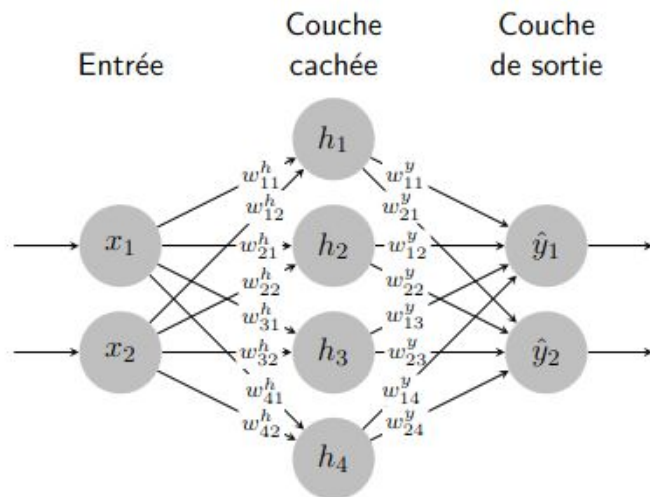
$$\nabla_{\theta} \mathcal{R}_{\theta}(\hat{B}) = \frac{1}{\#B} \sum_{k=0, \dots, B-1} \nabla_{\theta} l(f_{\theta}(x_k), y_k)$$

- Boils down to computing gradients for one sample

$$\nabla_{\theta} l(f_{\theta}(x), y)$$

$$l := l(f_{\theta}(x), y)$$

- Networks are complex but made of simple parts!
 - Simple gradients of component functions
 - Chain-rule allows decomposition into simple gradients $\frac{\partial l}{\partial w} = \frac{\partial l}{\partial a} \frac{\partial a}{\partial w}$
- Need to store intermediate activations “a” to evaluate partial derivatives $\frac{\partial a}{\partial w}$

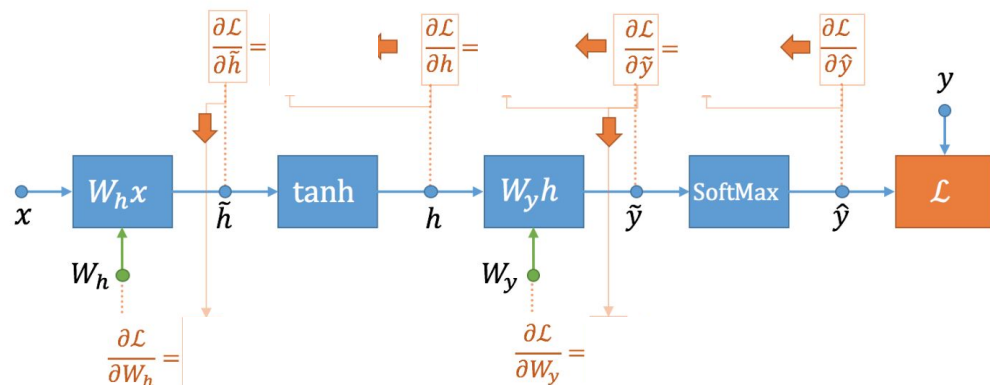
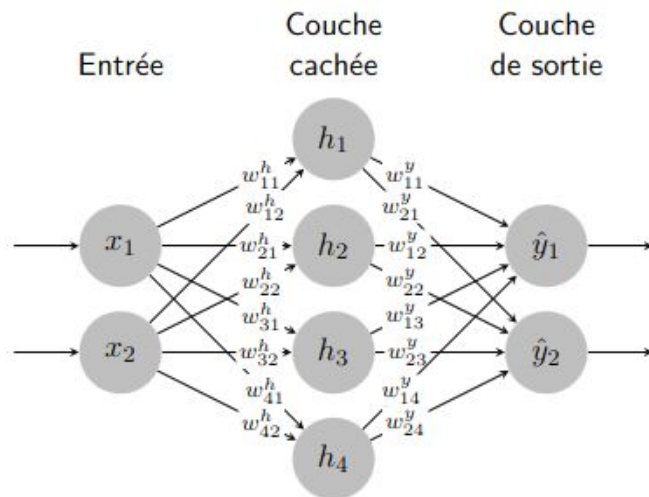


- Simple 1 hidden layer MLP
 - 2 inputs
 - 2 outputs
 - 4 hidden activations
- Classification problem
 - Outputs probabilities
 - Cross-entropy loss

$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$

Example: Tanh MLP

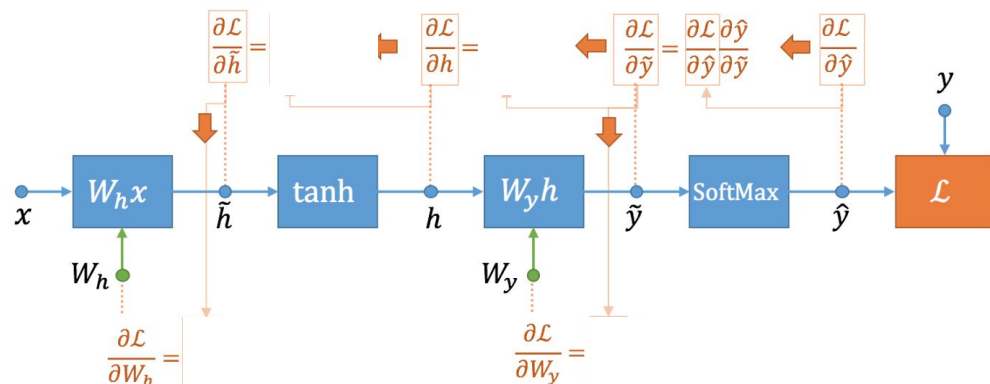
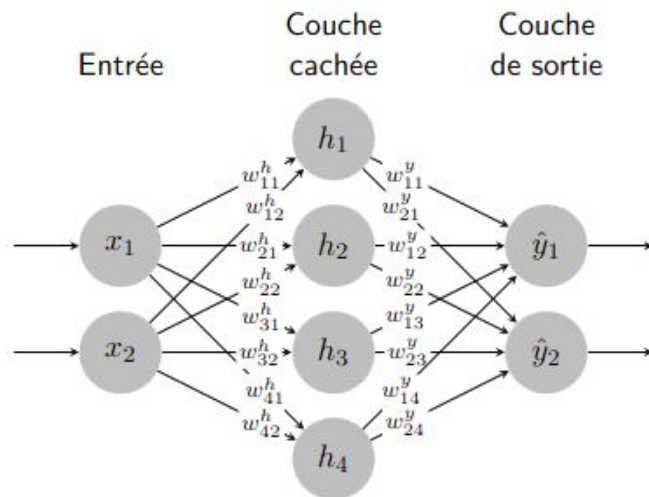
$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

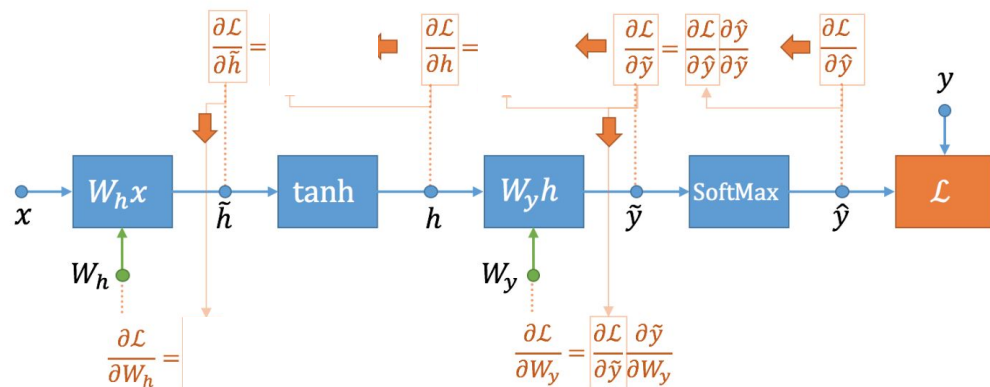
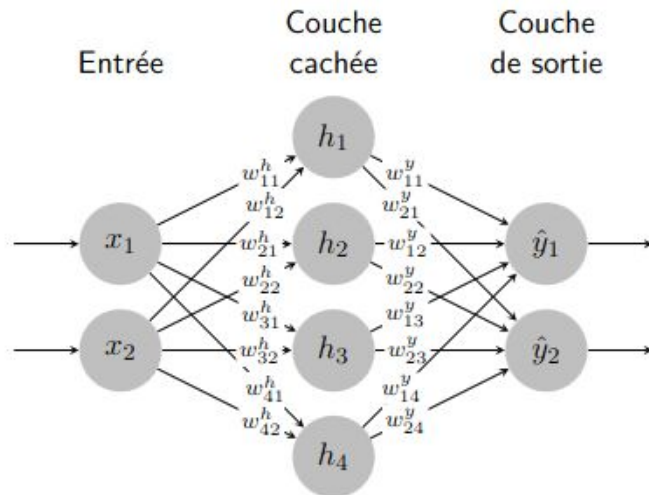
$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

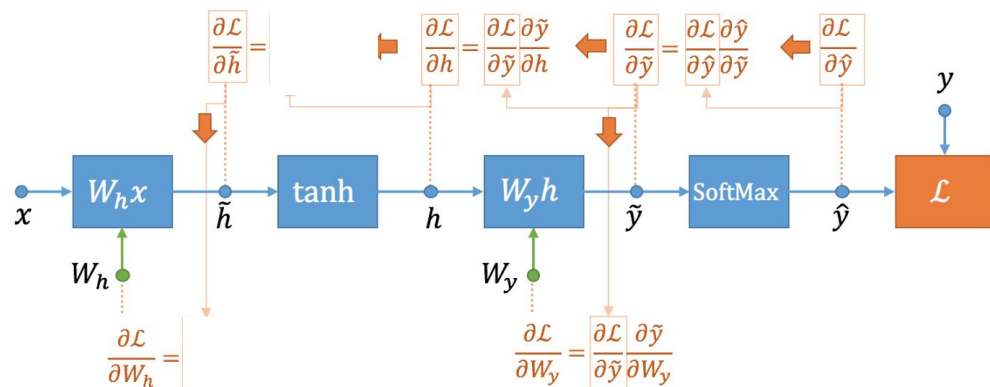
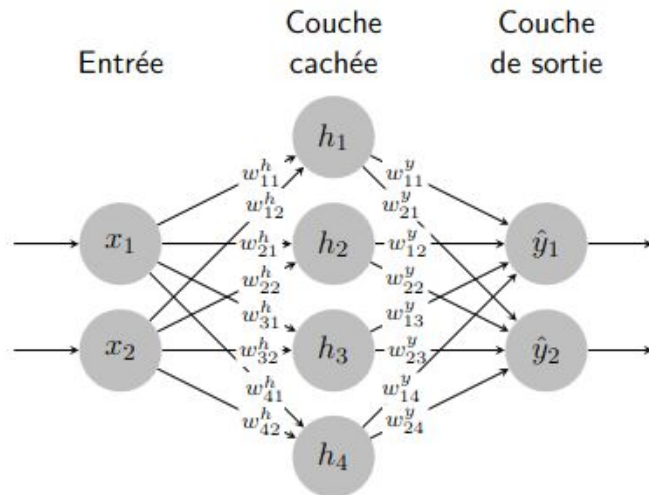
$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

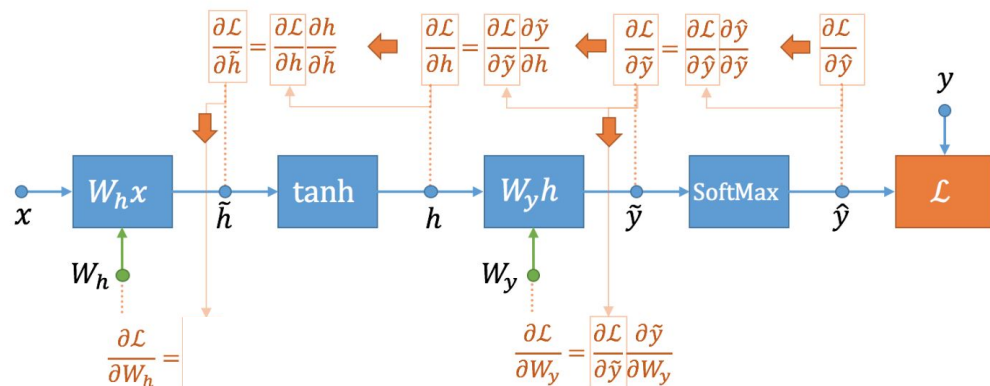
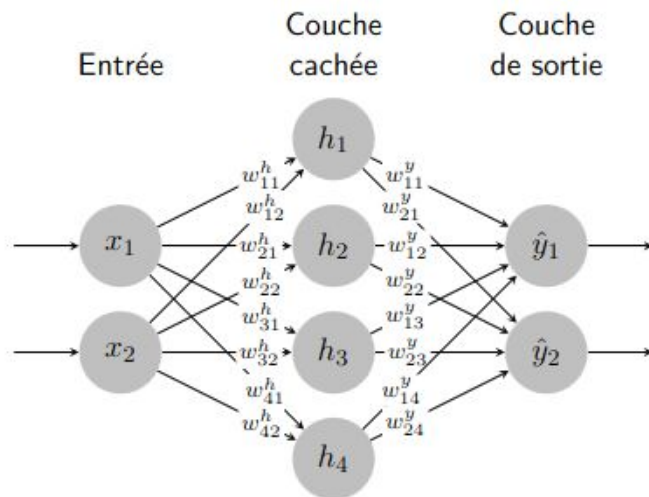
$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

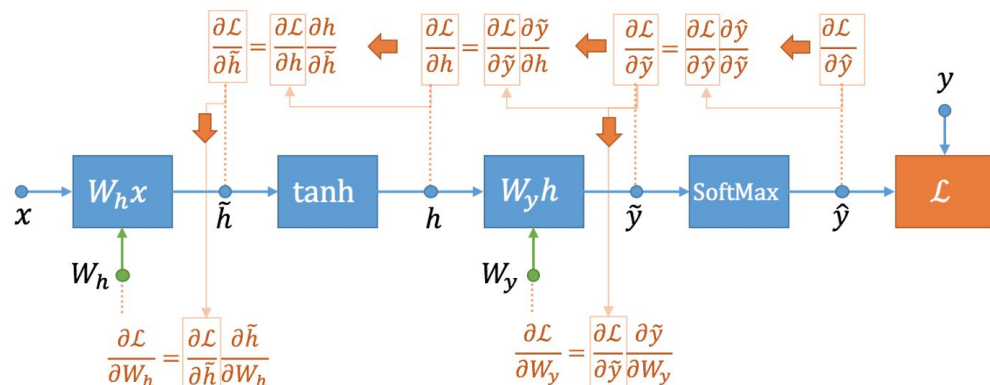
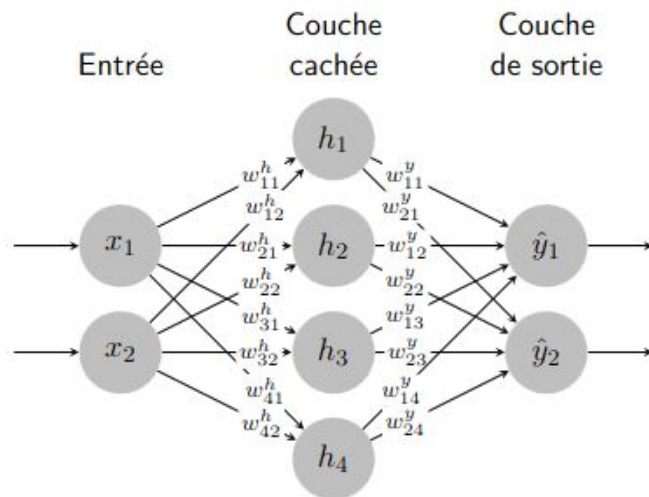
$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



$$\begin{cases} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{cases}$$

Example: Tanh MLP

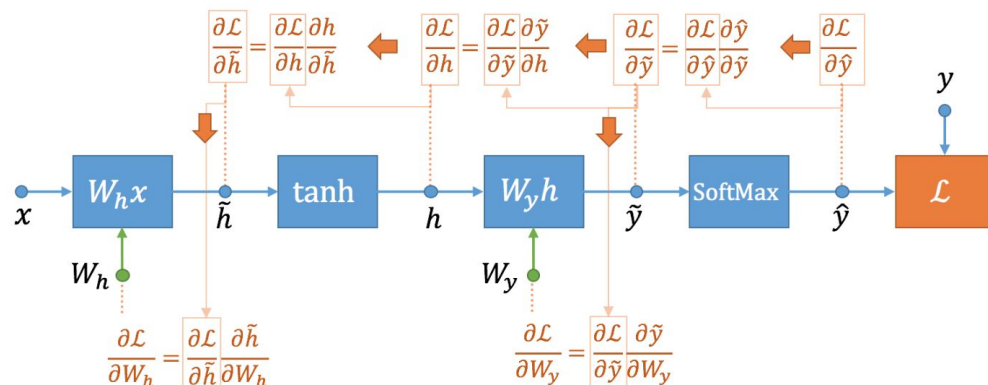
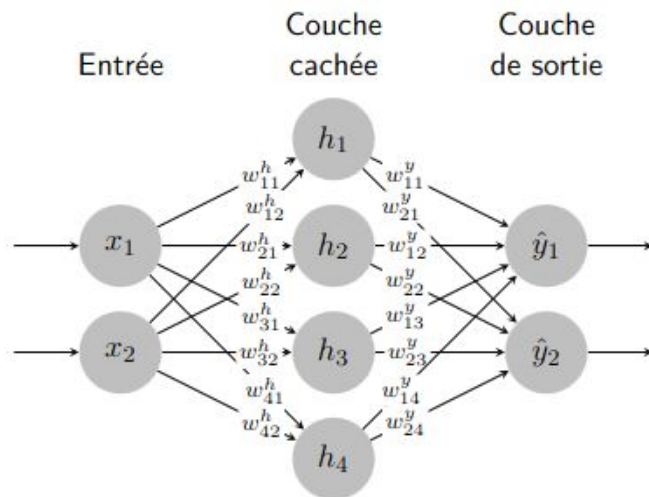
$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



$$\left\{ \begin{array}{l} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{array} \right\} \quad \left\{ \begin{array}{l} \delta_i^y = \frac{\partial \ell}{\partial \tilde{y}_i} = \\ \frac{\partial \ell}{\partial W_{i,j}^y} = \\ \frac{\partial \ell}{\partial b_i^y} = \end{array} \right.$$

Example: Tanh MLP

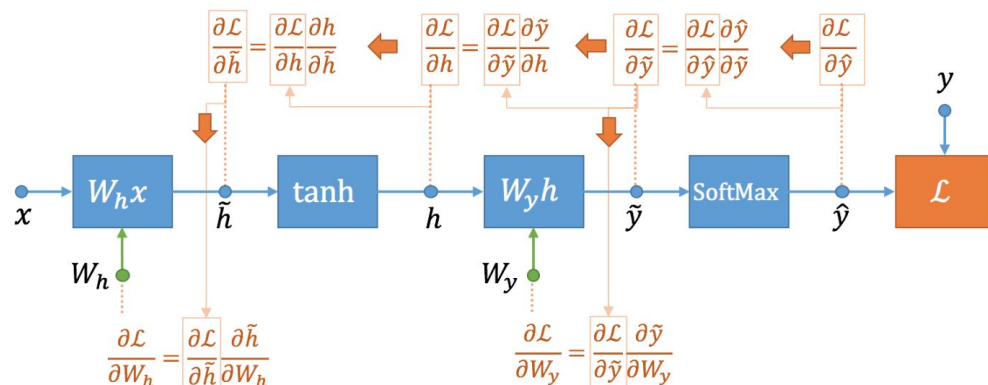
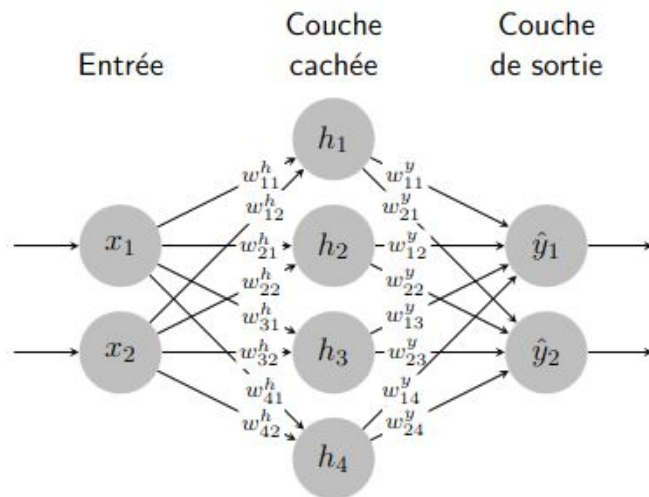
$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



$$\left\{ \begin{array}{l} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{array} \right\} \quad \left\{ \begin{array}{l} \delta_i^y = \frac{\partial \ell}{\partial \tilde{y}_i} = \hat{y}_i - y_i \\ \frac{\partial \ell}{\partial W_{i,j}^y} = \delta_i^y h_j \\ \frac{\partial \ell}{\partial b_i^y} = \delta_i^y \end{array} \right.$$

Example: Tanh MLP

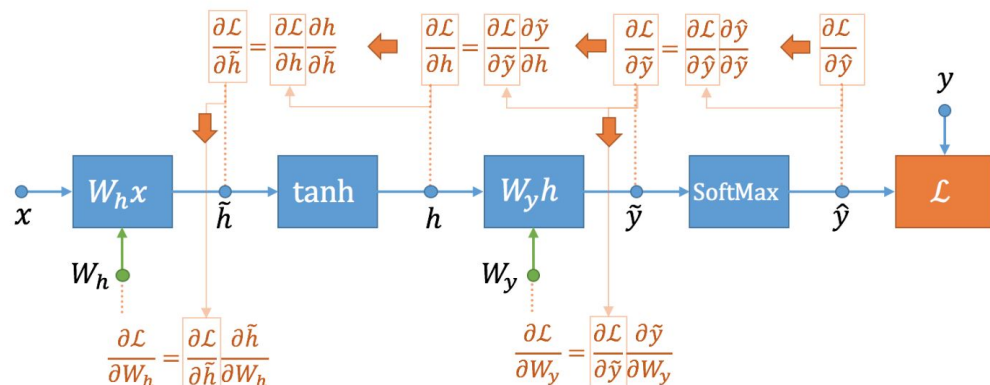
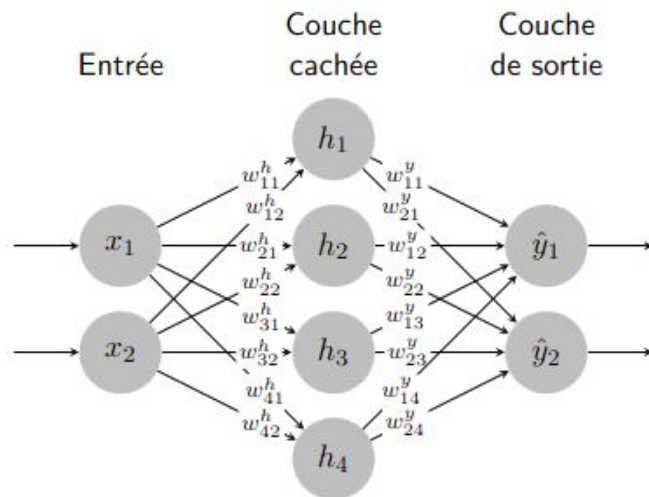
$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



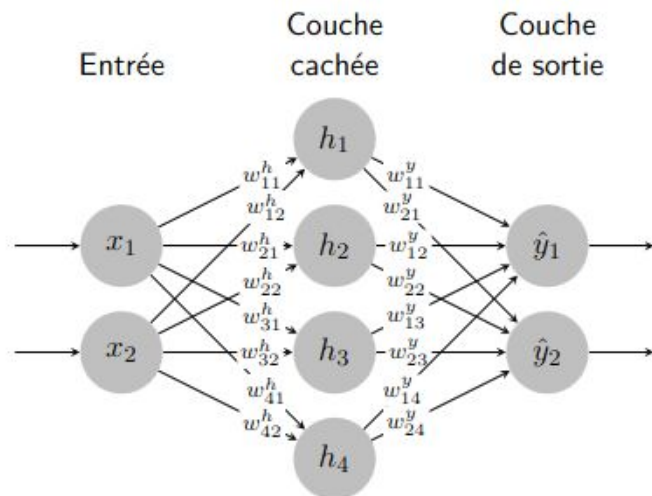
$$\left\{ \begin{array}{l} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{array} \right. \quad \left\{ \begin{array}{l} \delta_i^y = \frac{\partial \ell}{\partial \tilde{y}_i} = \hat{y}_i - y_i \\ \frac{\partial \ell}{\partial W_{i,j}^y} = \delta_i^y h_j \\ \frac{\partial \ell}{\partial b_i^y} = \delta_i^y \\ \delta_i^h = \frac{\partial \ell}{\partial \tilde{h}_i} = \\ \frac{\partial \ell}{\partial W_{i,j}^h} = \\ \frac{\partial \ell}{\partial b_i^h} = \end{array} \right.$$

Example: Tanh MLP

$$l_{CE}(\hat{y}, y) = - \sum_{i=0}^{\#Classes-1} y_i \log(\hat{y}_i)$$



$$\left\{ \begin{array}{l} \tilde{h}_i = \sum_{j=1}^{n_x} W_{i,j}^h x_j + b_i^h \\ h_i = \tanh(\tilde{h}_i) \\ \tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \\ \hat{y}_i = \text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \end{array} \right. \quad \left\{ \begin{array}{l} \delta_i^y = \frac{\partial \ell}{\partial \tilde{y}_i} = \hat{y}_i - y_i \\ \frac{\partial \ell}{\partial W_{i,j}^y} = \delta_i^y h_j \\ \frac{\partial \ell}{\partial b_i^y} = \delta_i^y \\ \delta_i^h = \frac{\partial \ell}{\partial \tilde{h}_i} = (1 - h_i^2) \sum_{j=1}^{n_y} \delta_j^y W_{j,i}^y \\ \frac{\partial \ell}{\partial W_{i,j}^h} = \delta_i^h x_j \\ \frac{\partial \ell}{\partial b_i^h} = \delta_i^h \end{array} \right.$$



$$\begin{cases} \tilde{\mathbf{h}} = \mathbf{x}\mathbf{W}^h + \mathbf{b}^h \\ \mathbf{h} = \tanh(\tilde{\mathbf{h}}) \\ \tilde{\mathbf{y}} = \mathbf{h}\mathbf{W}^y + \mathbf{b}^y \\ \hat{\mathbf{y}} = \text{SoftMax}(\tilde{\mathbf{y}}) \end{cases}$$

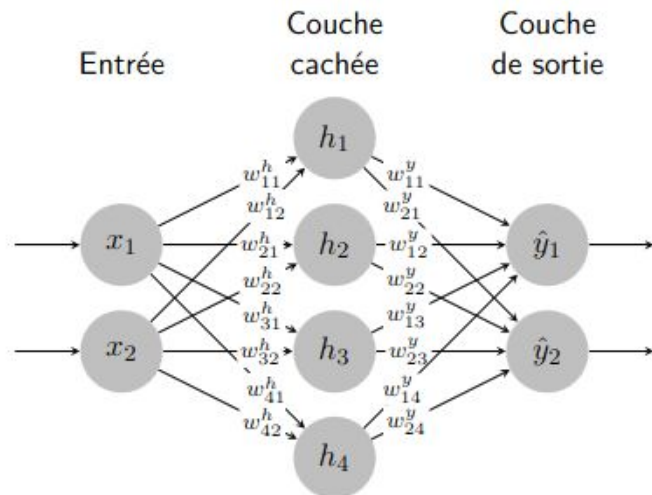
$$\begin{cases} \nabla_{\hat{\mathbf{y}}} = \hat{\mathbf{y}} - \mathbf{y} \\ \nabla_{\mathbf{W}^y} = \nabla_{\hat{\mathbf{y}}}^\top \mathbf{h} \\ \nabla_{\mathbf{b}^y} = \nabla_{\hat{\mathbf{y}}}^\top \\ \nabla_{\tilde{\mathbf{h}}} = (\nabla_{\hat{\mathbf{y}}} \mathbf{W}^y) \odot (1 - \mathbf{h}^2) \\ \nabla_{\mathbf{W}^h} = \nabla_{\tilde{\mathbf{h}}}^\top \mathbf{x} \\ \nabla_{\mathbf{b}^h} = \nabla_{\tilde{\mathbf{h}}}^\top \end{cases}$$

- Lecture 3 practical correction on Moodle
 - Implement this by hand with basic torch!
 - Careful with batch dimension!

```
1 def init_params(nx, nh, ny):
2     """
3     nx, nh, ny: integers
4     out params: dictionnary
5     """
6     params = {}
7
8     params["Wh"] = torch.randn((nh, nx))*0.3
9     params["Wy"] = torch.randn((ny, nh))*0.3
10    params["bh"] = torch.zeros((nh,1))
11    params["by"] = torch.zeros((ny,1))
12
13    return params
```

```
1 def forward(params, X):
2     """
3     params: dictionnary
4     X: (n_batch, dimension)
5     """
6     bsize = X.size(0)
7     nh = params['Wh'].size(0)
8     ny = params['Wy'].size(0)
9     outputs = {}
10
11     outputs["X"] = X
12     outputs["htilde"] = torch.mm(X, params["Wh"].T) + params["bh"].T
13     outputs["h"] = torch.tanh(outputs["htilde"])
14     outputs["ytilde"] = torch.mm(outputs["h"], params["Wy"].T) + params["by"].T
15     outputs["yhat"] = torch.exp(outputs["ytilde"])
16     outputs["yhat"] = outputs["yhat"] / outputs["yhat"].sum(dim=-1, keepdim=True)
17
18
19     return outputs['yhat'], outputs
```

```
1 def loss_accuracy(Yhat, Y):
2
3
4     L = - torch.mean((Y * torch.log(Yhat)).sum(dim=1)) # mean for the batch
5
6     _, indYhat = torch.max(Yhat, 1)
7     _, indY = torch.max(Y, 1)
8
9     acc = torch.sum(indY == indYhat) * 100. / indY.size(0);
10
11
12     return L, acc
```



$$\begin{cases} \tilde{\mathbf{h}} = \mathbf{x}\mathbf{W}^h + \mathbf{b}^h \\ \mathbf{h} = \tanh(\tilde{\mathbf{h}}) \\ \tilde{\mathbf{y}} = \mathbf{h}\mathbf{W}^y + \mathbf{b}^y \\ \hat{\mathbf{y}} = \text{SoftMax}(\tilde{\mathbf{y}}) \end{cases}$$

$$\begin{cases} \nabla_{\hat{\mathbf{y}}} = \hat{\mathbf{y}} - \mathbf{y} \\ \nabla_{\mathbf{W}^y} = \nabla_{\hat{\mathbf{y}}}^T \mathbf{h} \\ \nabla_{\mathbf{b}^y} = \nabla_{\hat{\mathbf{y}}}^T \\ \nabla_{\tilde{\mathbf{h}}} = (\nabla_{\hat{\mathbf{y}}} \mathbf{W}^y) \odot (1 - \mathbf{h}^2) \\ \nabla_{\mathbf{W}^h} = \nabla_{\tilde{\mathbf{h}}}^T \mathbf{x} \\ \nabla_{\mathbf{b}^h} = \nabla_{\tilde{\mathbf{h}}}^T \end{cases}$$

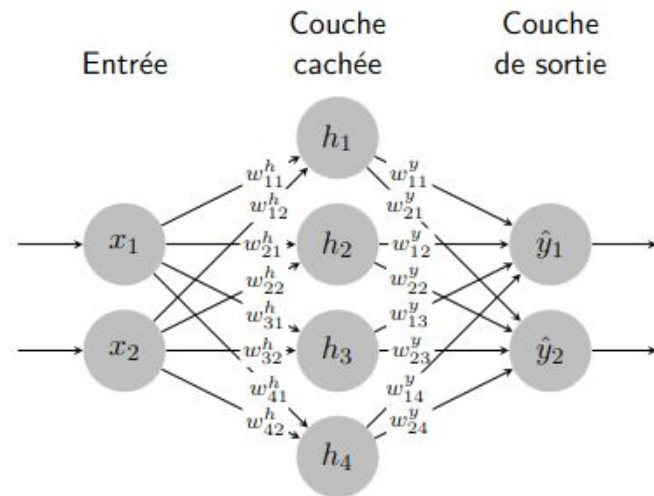
- Lecture 3 practical correction on Moodle
 - Implement this by hand with basic torch!
 - Careful with batch dimension!

```
1 def backward(params, outputs, Y):
2     bsize = Y.shape[0]
3     grads = {}
4
5     Y_tilde_grad = outputs["yhat"] - Y
6     h_tilde_grad = torch.mm(Y_tilde_grad, params['Wy']
7                             ) * (1 - torch.pow(outputs['h'], 2))
8
9     grads["Wy"] = torch.mm(Y_tilde_grad.T, outputs["h"])
10    grads["Wh"] = torch.mm(h_tilde_grad.T, outputs['X'])
11    grads["by"] = Y_tilde_grad.sum(dim=0, keepdim=True).T
12    grads["bh"] = h_tilde_grad.sum(0, keepdim=True).T
13
14    grads['Wy'] /= bsize
15    grads['by'] /= bsize
16    grads['Wh'] /= bsize
17    grads['bh'] /= bsize
18
19    return grads
```

```
1 def sgd(params, grads, eta):  
2  
3     params['Wy'] -= eta * grads['Wy']  
4     params['Wh'] -= eta * grads['Wh']  
5     params['by'] -= eta * grads['by']  
6     params['bh'] -= eta * grads['bh']  
7  
8     return params
```

```
for j in range(N // Nbatch):  
  
    indsBatch = range(j * Nbatch, (j+1) * Nbatch)  
    X = Xtrain[indsBatch, :]  
    Y = Ytrain[indsBatch, :]  
  
    Y_hat, outputs = forward(params, X)  
    loss, accuracy = loss_accuracy(Y_hat, Y)  
    grads = backward(params, outputs, Y)  
    params = sgd(params, grads, eta)
```


- Important to really know what is under the hood
 - Invisible in everyday pytorch/tf/jax use
 - Understand errors
 - Necessary to implement custom layers
 - Helps understand why it works

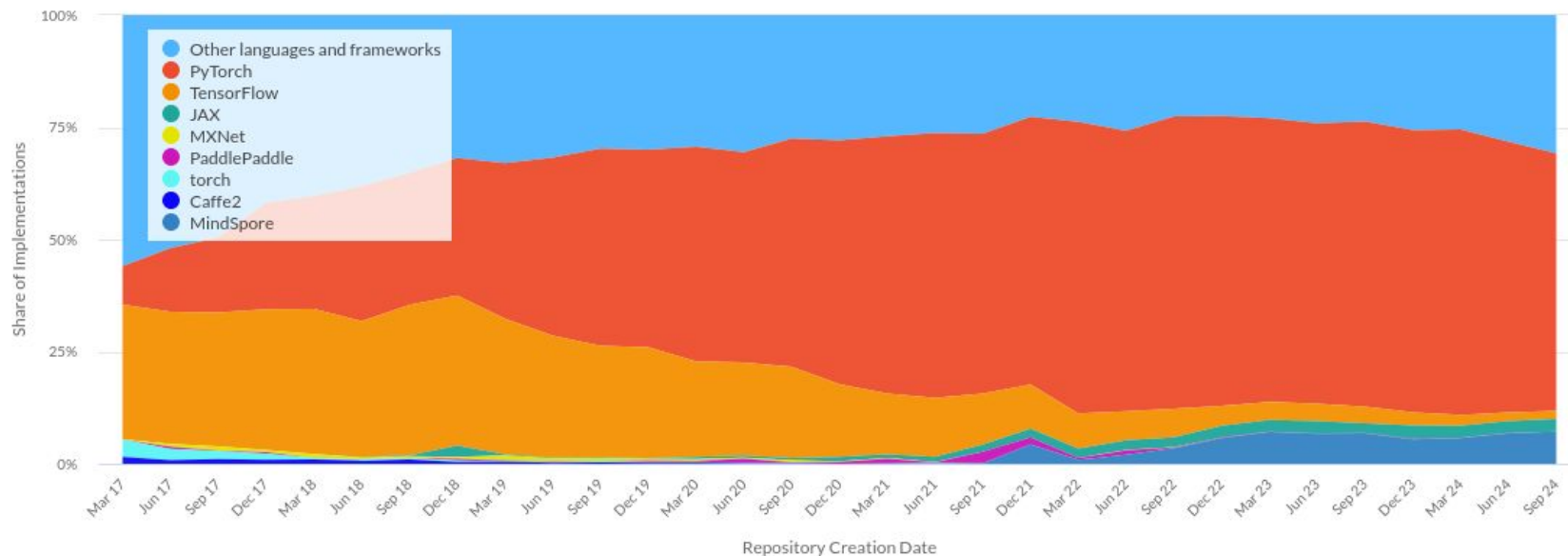


$$\begin{cases} \tilde{\mathbf{h}} = \mathbf{x}\mathbf{W}^h{}^\top + \mathbf{b}^h \\ \mathbf{h} = \tanh(\tilde{\mathbf{h}}) \\ \tilde{\mathbf{y}} = \mathbf{h}\mathbf{W}^y{}^\top + \mathbf{b}^y \\ \hat{\mathbf{y}} = \text{SoftMax}(\tilde{\mathbf{y}}) \end{cases}$$

- TP4a: Backprop, with actual tools
 - Manual -> fully automated
 - Understand how everything fits together
- TP4b: Computer Vision practical
 - Quick showcase of standard workflows
 - Practice standard loop from 4a
- TP4c: Natural Language Processing practical
 - Quick showcase of standard workflows
 - Practice standard loop from 4a

Pytorch vs other frameworks

Paper Implementations grouped by framework



```
def backward(params, outputs, Y):
    bsize = Y.shape[0]
    grads = {}

    Y_tilde_grad = outputs["yhat"] - Y
    h_tilde_grad = torch.mm(Y_tilde_grad, params['Wy'])
    h_tilde_grad = h_tilde_grad * (1 - torch.pow(outputs['h'], 2))

    grads["Wy"] = torch.mm(Y_tilde_grad.T, outputs["h"])
    grads["Wh"] = torch.mm(h_tilde_grad.T, outputs["X"])
    grads["by"] = Y_tilde_grad.sum(dim=0, keepdim=True).T
    grads["bh"] = h_tilde_grad.sum(0, keepdim=True).T

    grads['Wy'] /= bsize
    grads['by'] /= bsize
    grads['Wh'] /= bsize
    grads['bh'] /= bsize

    return grads
```

- Torch.tensor object
 - Np.array like
 - Tracked on a computational graph
 - .grad variable to track gradients
 - .backward to backpropagate gradients through the graph
 - Activate .autograd!

TP4a: Autograd backward

```
def backward(params, outputs, Y):
    bsize = Y.shape[0]
    grads = {}

    Y_tilde_grad = outputs["yhat"] - Y
    h_tilde_grad = torch.mm(Y_tilde_grad, params['Wy'])
    h_tilde_grad = h_tilde_grad * (1 - torch.pow(outputs['h'], 2))

    grads["Wy"] = torch.mm(Y_tilde_grad.T, outputs["h"])
    grads["Wh"] = torch.mm(h_tilde_grad.T, outputs["X"])
    grads["by"] = Y_tilde_grad.sum(dim=0, keepdim=True).T
    grads["bh"] = h_tilde_grad.sum(0, keepdim=True).T

    grads['Wy'] /= bsize
    grads['by'] /= bsize
    grads['Wh'] /= bsize
    grads['bh'] /= bsize

    return grads
```

```
params['Wh'] = torch.randn(nh, nx) * 0.3
params['Wh'].requires_grad = True
params['bh'] = torch.zeros(nh, 1, requires_grad=True)
params['Wy'] = torch.randn(ny, nh) * 0.3
params['Wy'].requires_grad = True
params['by'] = torch.zeros(ny, 1, requires_grad=True)
```

```
with torch.no_grad():
    params['Wy'] -= eta * params['Wy'].grad
    params['Wh'] -= eta * params['Wh'].grad
    params['by'] -= eta * params['by'].grad
    params['bh'] -= eta * params['bh'].grad

    params['Wy'].grad.zero_()
    params['Wh'].grad.zero_()
    params['by'].grad.zero_()
    params['bh'].grad.zero_()
```

```
yhat, outputs = forward(params, X)
L, acc = loss_accuracy(yhat, Y)
L.backward()
params = sgd(params, eta)
```

```
params = {}

params["Wh"] = torch.randn((nh, nx))*0.3
params["Wy"] = torch.randn((ny, nh))*0.3
params["bh"] = torch.zeros((nh,1))
params["by"] = torch.zeros((ny,1))
```

```
def forward(params, X):
    """
    params: dictionary
    X: (n_batch, dimension)
    """
    bsize = X.size(0)
    nh = params['Wh'].size(0)
    ny = params['Wy'].size(0)
    outputs = {}

    outputs["X"] = X
    outputs["htilde"] = torch.mm(X, params["Wh"].T) + params["bh"].T
    outputs["h"] = torch.tanh(outputs["htilde"])
    outputs["ytilde"] = torch.mm(outputs["h"], params["Wy"].T) + params["by"].T
    outputs["yhat"] = torch.exp(outputs["ytilde"])
    outputs["yhat"] = outputs["yhat"] / outputs["yhat"].sum(dim=-1, keepdim=True)

    return outputs['yhat'], outputs
```

- Torch.nn.Module objects
 - `__init__` creates weights and initializes them!
 - `.forward` implements forward operations
 - Default object call
 - `model(x)`
 - Some global control over model weights

```
params = {}
```

```
params["Wh"] = torch.randn((nh, nx))*0.3  
params["Wy"] = torch.randn((ny, nh))*0.3  
params["bh"] = torch.zeros((nh,1))  
params["by"] = torch.zeros((ny,1))
```

```
def forward(params, X):  
    """  
    params: dictionnaire  
    X: (n_batch, dimension)  
    """  
    bsize = X.size(0)  
    nh = params["Wh"].size(0)  
    ny = params["Wy"].size(0)  
    outputs = {}  
  
    outputs["X"] = X  
    outputs["htilde"] = torch.mm(X, params["Wh"].T) + params["bh"].T  
    outputs["h"] = torch.tanh(outputs["htilde"])  
    outputs["ytilde"] = torch.mm(outputs["h"], params["Wy"].T) + params["by"].T  
    outputs["yhat"] = torch.exp(outputs["ytilde"])  
    outputs["yhat"] = outputs["yhat"] / outputs["yhat"].sum(dim=-1, keepdim=True)  
  
    return outputs['yhat'], outputs
```

```
model = torch.nn.Sequential(  
    torch.nn.Linear(nx, nh),  
    torch.nn.Tanh(),  
    torch.nn.Linear(nh, ny)  
)  
loss = torch.nn.CrossEntropyLoss()
```

```
_, indY = torch.max(Y, 1)  
L = loss(Yhat, indY)  
  
_, indYhat = torch.max(Yhat, 1)  
acc = torch.sum(indY == indYhat.data) * 100 // indY.size(0);
```

```
with torch.no_grad():  
    for param in model.parameters():  
        param -= eta * param.grad  
    model.zero_grad()
```

```
yhat = model(X)  
L, acc = loss_accuracy(loss, yhat, Y)  
L.backward()  
model = sgd(model, eta)
```



```
def sgd(params, grads, eta):  
  
    params['Wy'] -= eta * grads['Wy']  
    params['Wh'] -= eta * grads['Wh']  
    params['by'] -= eta * grads['by']  
    params['bh'] -= eta * grads['bh']  
  
    return params
```

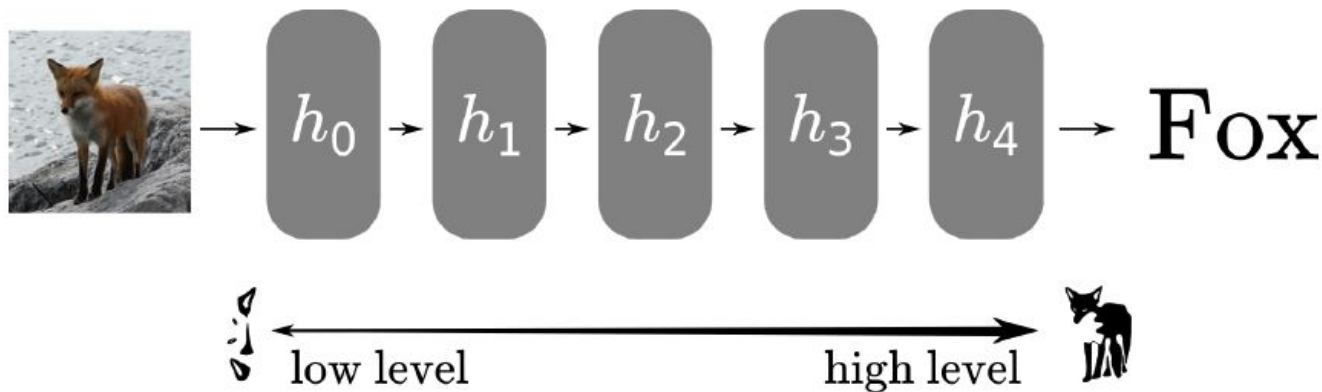
- Torch.optim objects
 - Tracks learning rates
 - Tracks weights to optimize
 - Performs SGD steps
 - Even cleans up!


```
def sgd(params, grads, eta):  
  
    params['Wy'] -= eta * grads['Wy']  
    params['Wh'] -= eta * grads['Wh']  
    params['by'] -= eta * grads['by']  
    params['bh'] -= eta * grads['bh']  
  
    return params
```

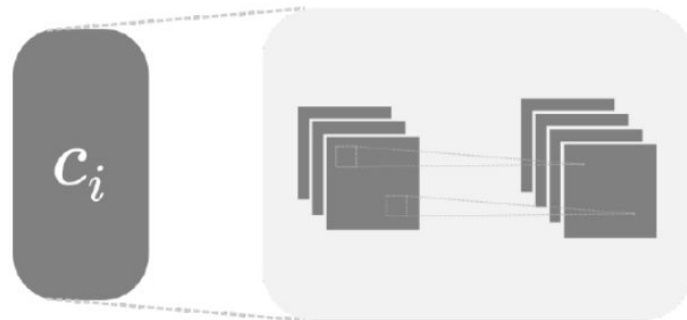
```
optim = torch.optim.SGD(model.parameters(), lr=eta)
```

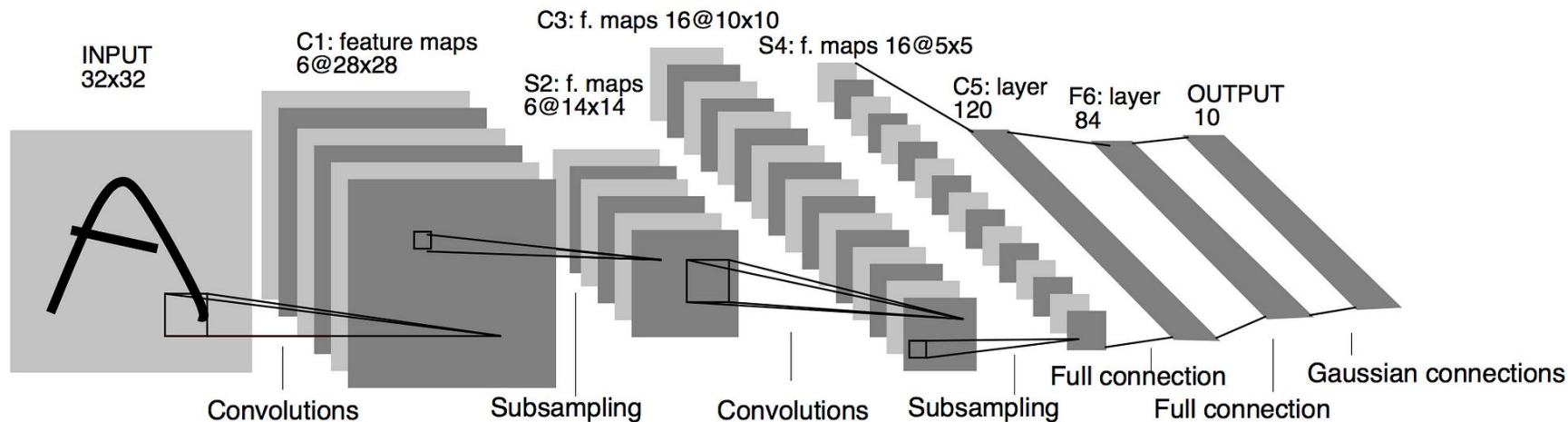
```
yhat = model(X)  
L, acc = loss_accuracy(loss, yhat, Y)  
optim.zero_grad()  
L.backward()  
optim.step()
```

- Training a network requires
 - Weights
 - A forward function
 - A backward function
 - Gradient steps
- Nice pytorch tools
 - `Torch.tensor` and `torch.autograd`
 - `Torch.nn`
 - `Torch.optim`



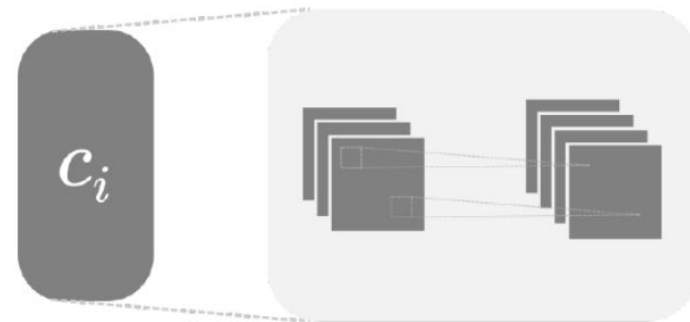
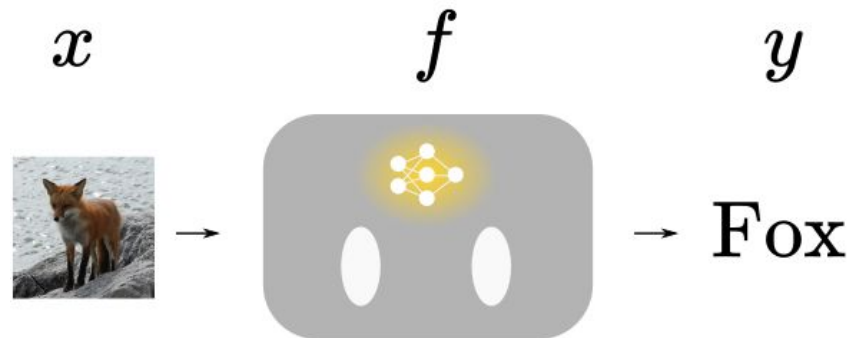
- Convolutional layers
 - Local correlations
 - Well suited to images
 - Used sometimes with Transformers now



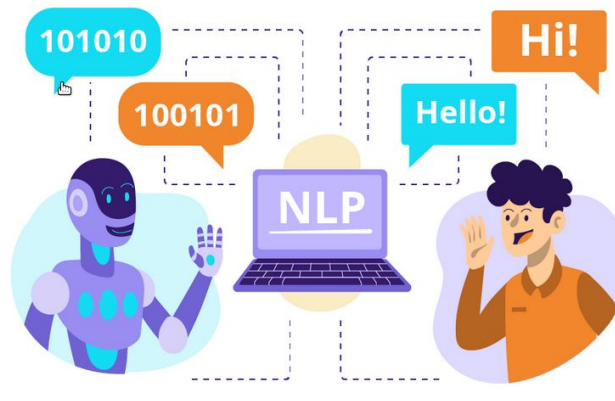


- Classical architecture of computer vision
 - Convolutional layers for feature extraction
 - Dense/linear layers to make decisions from features
 - E.g. LeNet5 (Before 2000!)

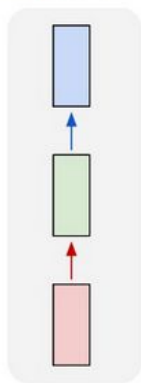
- A few milestones
 - Load image data
 - Load a classic model
 - Train it!
 - (Finetune a strong model)
- Apply knowledge from TP4a!



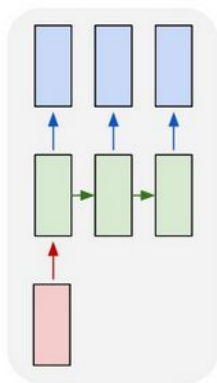
- Recurrent networks
 - Temporal correlations
 - How to take the past into account?
 - Recent resurgence (SSMs, ...)



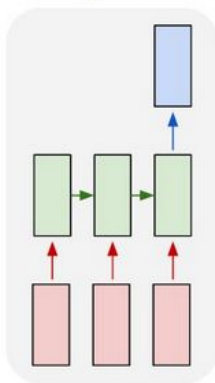
one to one



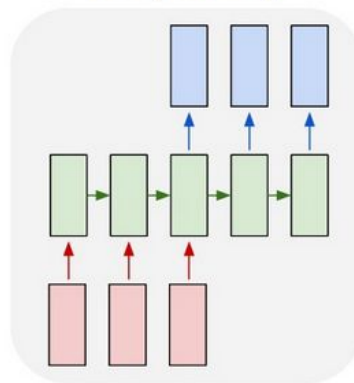
one to many



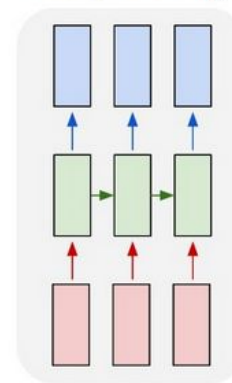
many to one

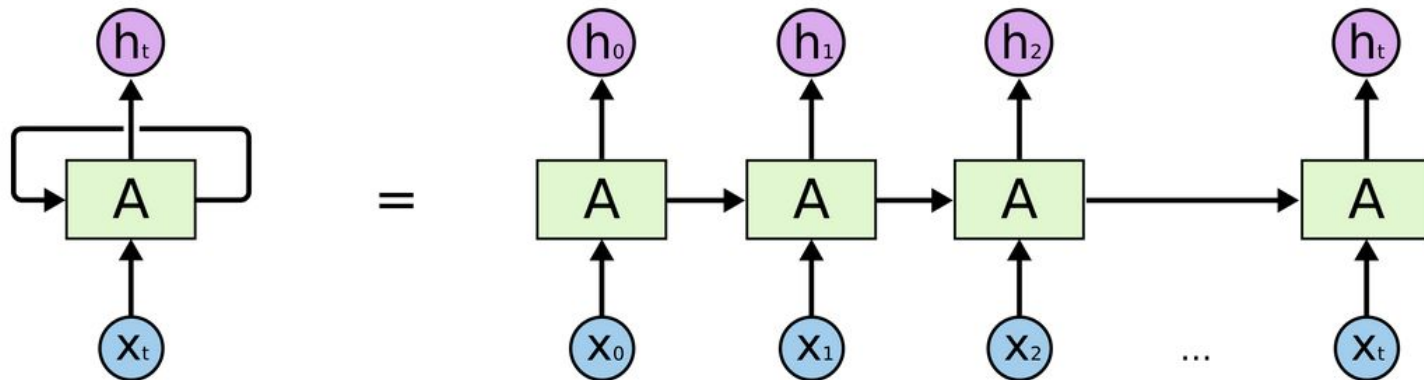


many to many

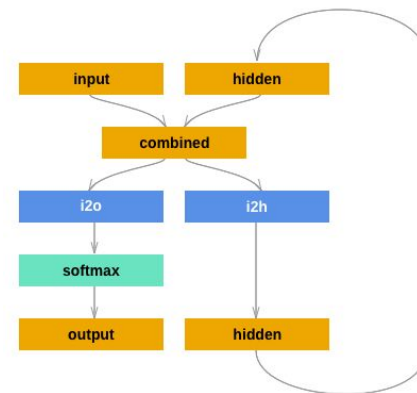


many to many





- Basic RNN model
 - Input + hidden state
 - Hidden state remains from input to input



- Making a language processor
 - Tokenize words
 - Create network
 - Train network
 - Apply network
- Apply knowledge from TP4a!

