Deep Learning

Deep neural networks: optimization and regularization

Alexandre Fournier Montgieux

M2 BDMA, CentraleSupélec, Université Paris Saclay

October 2, 2024

Plan

Reminder scalar product (also called dot product and inner product)

A function $< .|. >$ of $E \times E \to \mathbb{R}$ is a scalar product if :

- it is symmetric i.e. : $\forall x, y \in E, < x|y >=< y|x >$
- it is bilinear i.e. linear w.r. each side :
  $\forall w, x, y, z \in E and \forall \lambda, \mu \in \mathbb{R}:< x|y + \lambda z >=< x|y > +\lambda < x|z >$ and
  $< x + \mu w|y >=< x|y > +\mu < w|y >$
- positive definite: $\forall x \in E* :< x|x >> 0$

For instance in $\mathbb{R}^n < x|y >= x^T y$ is a scalar product. For function $\mathbb{R} \to \mathbb{R}, < x|y >= \int_a^b f(x)g(x)dx$ is a scalar product.

Backpropagating gradients

- **Reminder**: learning of a NN is based on optimizing a cost function $J(\theta)$
- In practice, optimization performed by gradient descent $\rightarrow$we must be able to compute $\nabla_\theta J(\theta)$.
  - Problem: computationally very costly
- **Back-propagation** is an efficient gradient computation technique
  - **Not** a learning algorithm/training methodology
  - **Not** necessarily exclusive to neural networks

Chain rule

- The derivative of composition of functions is called the *chain rule* of calculus.
- If $y = g(x)$ and $z = f(y) = f(g(x)) = (f \circ g)(x)$:

$$\frac{df}{dx} = f'(g(x))g'(x) \quad \text{or} \quad \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- multivariate function (vector calculus):

$$\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n, g : \mathbb{R}^m \mapsto \mathbb{R}^n \text{ and } f : \mathbb{R}^n \mapsto \mathbb{R}$$

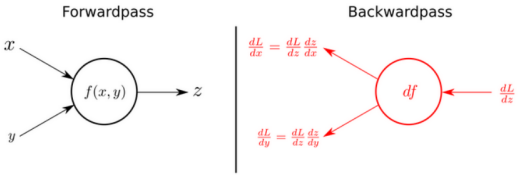
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

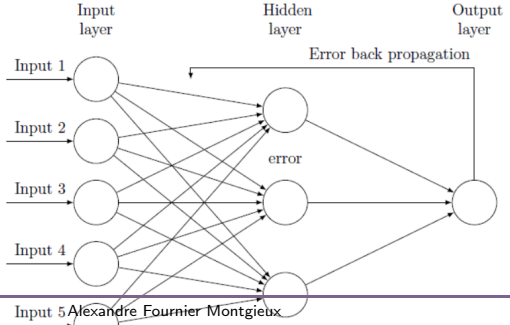
$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$$

where $\left(\frac{\partial y}{\partial x}\right)$ is the Jacobian (n x m) of $g()$.

Back-propagation

- Backpropagation is a recursive application of the chain rule from the cost function
  - Forward pass compute the output of the network
  - Cost function computes the error between expected output and actual output of the network
  - Backpropagation evaluates individual gradients of each parameter and propagates them backward to update them

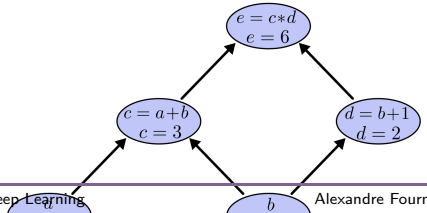




Forward-pass

- A feedforward network takes as input x and outputs $\hat{y}$
- Information flows from layer to layer ("forward propagation")

$$\mathbf{h}^{(1)} = g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}^{(1)} + \mathbf{b}^{(1)}\right)$$
$$\mathbf{h}^{(2)} = g^{(2)}\left(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right)$$
$$\dots$$
$$\hat{\mathbf{y}} = g^{(d)}\left(\mathbf{W}^{(d)}\mathbf{h}^{(d-1)} + \mathbf{b}^{(d)}\right)$$

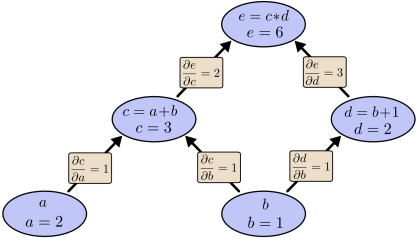


Derivatives in computational graph

- Local derivates of connected nodes are computed locally on the edges of the graph
- For non-connected nodes
  - product of edges connected between the nodes
  - sum over all paths

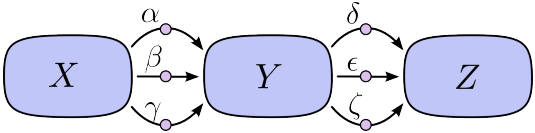
Just application of chain rule !

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 2 \times 1 + 3 \times 1 = 5$$

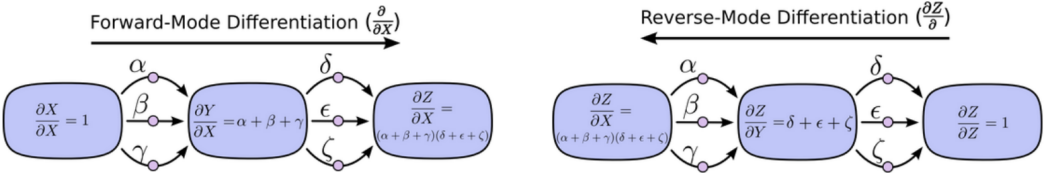


Derivatives in computational graph

- **Problem**: summing over all paths can quickly become computationally intractable
  - $\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$

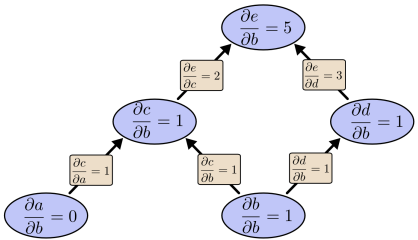


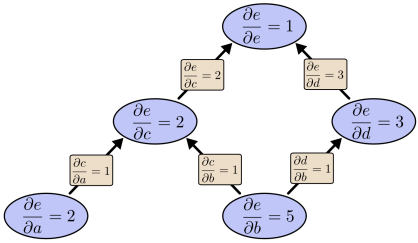
- Simplication by factorization: $\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$
  - 2 "modes", both doing only 1 path per node.



Why backpropagation?

- Which propagation mode to choose?





- reverse-mode differentiation (right) allows to obtain the derivative of the output with respect to **every** node directly in one pass $\rightarrow$**MASSIVE** parallelization (10 millions + in practice)
  - forward-mode differentiation only brings the derivative of one node with respect to one input during one pass

Reminder: why optimization?

- Neural networks = parametric model f estimating $p(\mathbf{y} \mid \mathbf{x}; \theta)$
- Maximum likelihood principle implies minimizing the following cost function:

$$J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{\text{data}}} \mathcal{L}(f(\mathbf{x}, \theta), y) = \mathbb{E}_{x,y\sim\hat{p}_{\text{data}}} \left[\log p_{\text{model}}\left(y \mid x\right)\right]$$

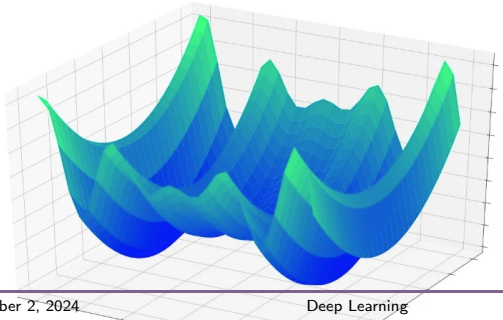
- Exact form of $J(\theta)$ depends on p_{model} $\rightarrow$depends on task/modelisation $\rightarrow$depends on output form
- in practice, we minimize the **empirical risk** over the training dataset (m samples):

$$\mathbb{E}_{x,y\sim\hat{p}_{\text{data}}} \mathcal{L}(f(\mathbf{x}, \theta), y) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(f\left(x_i; \theta\right), y_i\right)$$

- minimization by gradient descent: $\theta_j \leftarrow \theta_j - \epsilon \nabla_{\theta_j} \mathcal{L}(f(\mathbf{x}, \theta), y)$
- performance measured on a separate test dataset $\rightarrow$**generalization**

Optimization versus learning

- Learning as an **indirect** relationship to what we want to optimize:
  - We are interested in performance P on the **test set** in the end, which is sometimes untractable
  - We minimize a cost function $J(\theta$ **hoping** that it improves performance P on the test set
  - In "pure" optimization, minimizing $J(\theta$ is the true objective.



- In (machine) learning, we are mostly interested in **generalization** $\rightarrow$test performance (untractable)
- better local minimum doesn't guarantee better generalization

Problems of gradient-descent

- local minima and saddle points
  - small gradient $\rightarrow$can stop or greatly slow down gradient descent
- partial estimation of gradient slows down descent $\rightarrow$but can be beneficial for generalization

Problems of deep neural networks

- **Bad convergence**: many local optima
- **Long training time**: speed of SGD depends on initialization
- **Overfitting**: DNN have a lot of free parameters $\rightarrow$they tend to learn by heart the training set
- **Vanishing gradients**: first layers may not receive sufficient gradients early in training

Solving bad convergence

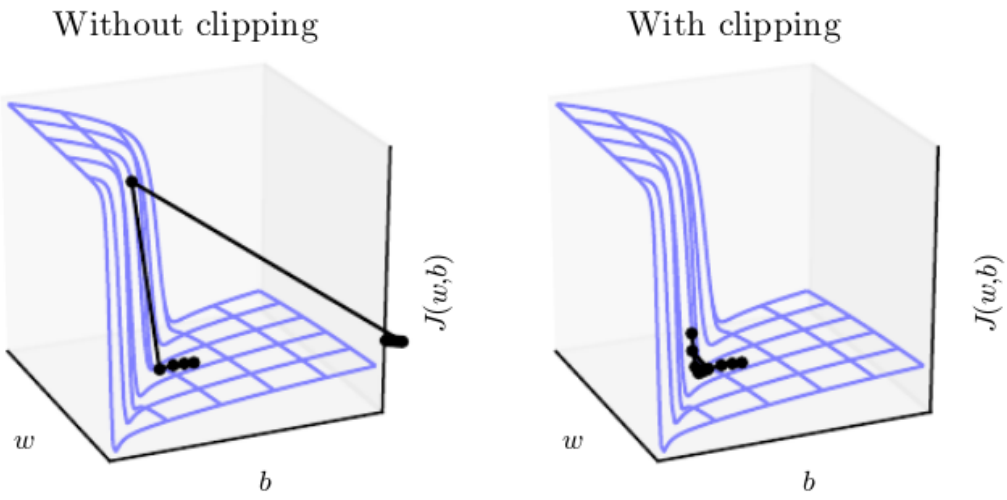
We need to stabilize and improve convergence of gradient descent on deep neural networks $\rightarrow$good optimization techniques.

Optimization techniques:

- Gradient clipping
- Optimizers

We need good generalization $\rightarrow$regularization.

Gradient clipping



Gradient clipping

- Solution: clip the gradient norm to a threshold v

$$\text{If} \|\mathbf{g}\| > v$$
$$\mathbf{g} \leftarrow \frac{\mathbf{g}v}{\|\mathbf{g}\|}$$

- keeps the direction
- prevents "overshooting"
- minibatch SGD gives an **unbiased** estimation of the gradient
  - gradient clipping is biased
  - but works well in practice !
  - Theoretical work exist to explain it

Momentum

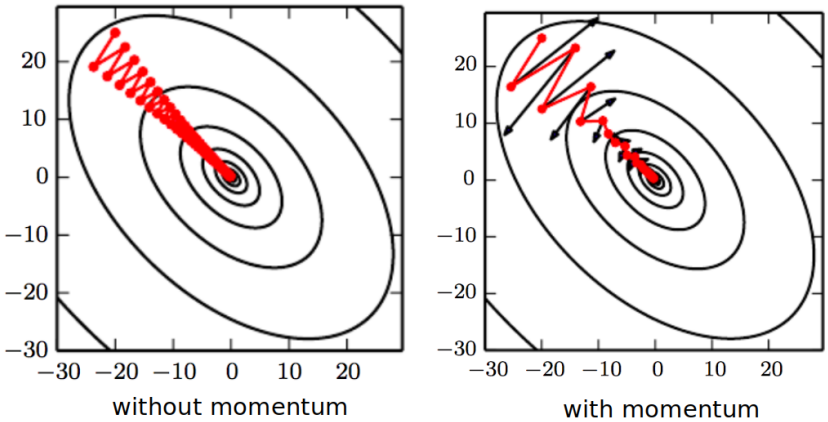
- Acceleration method for SGD
- Idea: smooth gradient steps with **momentum/inertia** →uses previous gradient steps as a "memory" of the direction

Let's define the velocity $v(\theta)$: direction + speed of parameters. For classic SGD, $v(\theta) = \nabla_{\theta_j}\mathcal{L}$. Let's denote the gradient as $G(\theta)$ to simplify the notations.

$$\begin{cases} v(\theta)^t = \alpha v(\theta)^{t-1} - (1-\alpha)\nabla_\theta \mathcal{L}(\theta) \\ \Delta\theta^t = -\eta v(\theta) \\ \theta^{t+1} = \theta^t + \Delta\theta^t \end{cases}$$

$0 \leq \alpha < 1$ controls how much of the gradient we use for the parameter update (usually around 0.9).$\alpha = 0$ is the vanilla SGD.

Momentum SGD



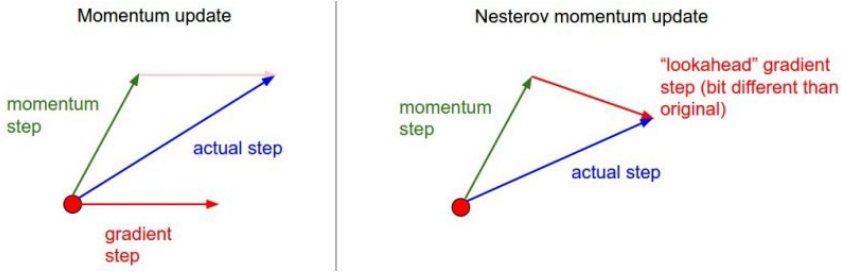
Visualization

SGD with Nesterov Momentum

Variant of momentum SGD

- SGD with momemtum tends to oscillate around the minimum
- Nesterov corrects the oscillations by estimating the gradient **after** the momemtum update:

$$v(\theta) = \alpha v(\theta) - (1-\alpha)\nabla_\theta \mathcal{L}(\theta + \alpha v(\theta))$$
$$\Delta\theta = \eta v(\theta)$$



Learning rate

Stochastic gradient descent (SGD) estimates gradient with only one sample and iterates over the whole dataset:

$$\theta \leftarrow \theta - \epsilon \nabla_\theta f\left(\theta; x^{(i)}, y^{(i)}\right)$$

We usually use a *minibatch* instead:

$$\theta \leftarrow \theta - \epsilon \nabla_\theta f\left(\theta; x^{(i:i+B)}, y^{(i:i+B)}\right) = \theta - \frac{\epsilon}{B} \nabla_\theta \sum_{i=i_0}^{i_0+B} L\left(x^{(i)}, y^{(i)}, \theta\right)$$

- Learning rate is one of the most important hyperparameter
- Realistically, it should decrease during training (as we get closer to the optimum).

Heuristics for learning rate

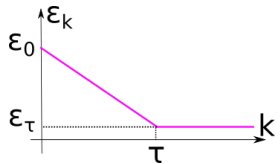
Why decrease the learning rate?

- True gradient becomes small with θ comes close to the minimum.
- With SGD, estimating the gradient with samples introduces "noise" $\rightarrow$estimated gradient doesn't nessecarily decreases !
- Sufficient condition for convergence of SGD is:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \quad \text{et} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

In practice

- Linear decay until τ: $\epsilon_k = \left(1 - \frac{k}{\tau}\right) \epsilon_0 + \frac{k}{\tau} \epsilon_\tau$
- Constant after τ



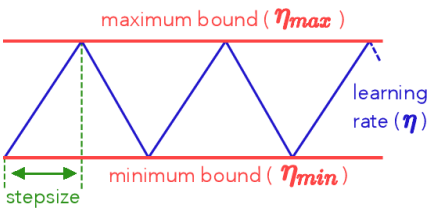
Heuristics for learning rate

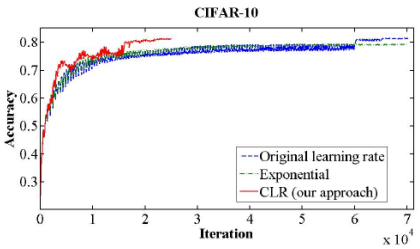
How to choose $\epsilon_0, \epsilon_\tau$ and τ ?

- With trial and error (train/validation)
- It is better to **monitor** the loss function during training
- Practical choice:
  - τ so that whole training dataset is seen $\sim$ 100 times (*epochs*)
  - $\epsilon_\tau \sim 1\%$ of ϵ_0
  - ϵ_0: comes with experience and "nose"
    - too big: big variations in loss
    - too small: learning is slow, loss can get stuck in a high plateau
  - Recipe:
    - try multiple ϵ_0 over 100 iterations
    - take ϵ_0 slightly higher than the best

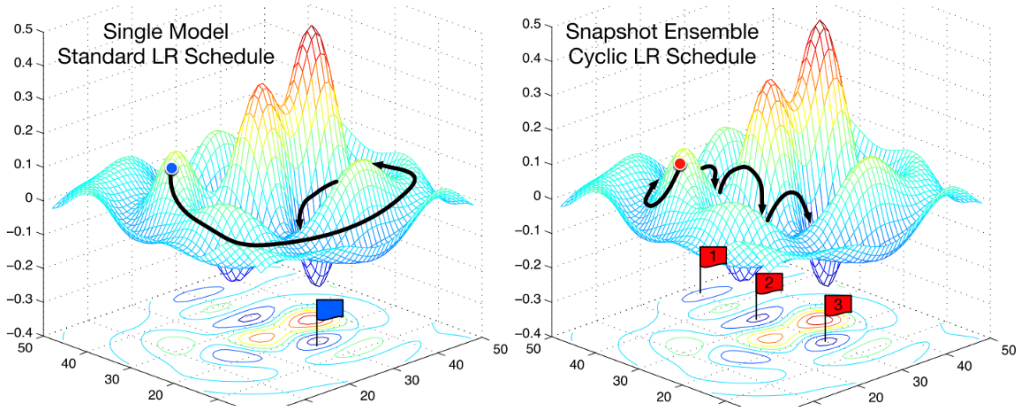
Cyclical learning rate

- Optimization difficulties comes more from "plateau" than bad local optima
- Increasing τ allows to go across these "plateaux"





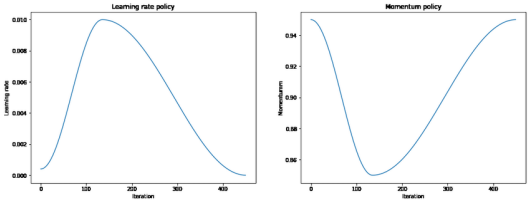
Cyclical learning rate

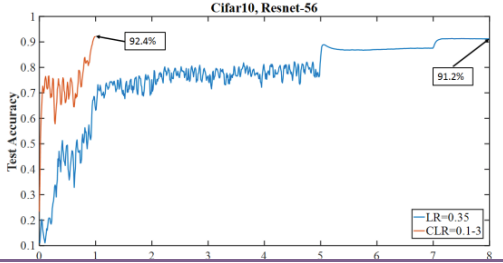


Cyclical learning rate allows to jump from local optimum to local optimum until we find a better one.

Super convergence with one-cycle policy

- Only one cycle
  - lr starts small to begin convergence
  - lr increases then stabilizes to high value to cross "big *plateaux*"
  - lr decreases to a small value to optimize local minima





SGD with adaptive learning rates

Preconditioning:

$$\theta^{t+1} \leftarrow \theta^t - \eta_t \mathbf{P}_t^{-1} G\left(\theta^t\right)$$

AdaGrad: Parameters with largest partial derivatives should have a rapid decrease.

$$\mathbf{P}_t = \left\{ \operatorname{diag}\left(\sum_{j=0}^{t} G\left(\theta^t\right) G\left(\theta^t\right)^\top \right) \right\}^{1/2}$$

RMSProp: Introduces momentum when computing the preconditioner.

$$\mathbf{P}_t = \left\{ \operatorname{diag}\left(\alpha \mathbf{P}_{t-1} + \sum_{j=0}^{t} G\left(\theta^t\right) G\left(\theta^t\right)^\top \right) \right\}^{1/2}$$

AdaGrad

AdaGrad: Parameters with largest partial derivatives should have a rapid decrease.

$$\begin{cases} \mathbf{g}^t \leftarrow G\left(\theta^t\right) \\ \mathbf{r}^t \leftarrow \mathbf{r}^{t-1} + \mathbf{g}^t \odot \mathbf{g}^t \\ \Delta\theta^t \leftarrow -\frac{\lambda}{\delta+\sqrt{\mathbf{r}^t}} \odot \mathbf{g}^t \\ \theta^{t+1} \leftarrow \theta^t + \Delta\theta^t \end{cases}$$

John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: JMLR(2011).

RMSProp

- AdaGrad rapidly converges on convex problems
- Keep all the history with momentum can be detrimentary + smoothing gradient destroys information
  - Start is convex then non convex
- **RMSProp** (Root Mean Square Propagation): adapt the learning rate to **curvature** of the loss function
  - put the **brakes** on when the function is steep (high gradient)
  - **accelerate** when the loss function is flat (plateau)

$$v(\theta) = \alpha v(\theta) + (1-\alpha)\left(\nabla_\theta \mathcal{L}(\theta)\right)^2$$

$$\Delta\theta = -\frac{\eta}{\epsilon + \sqrt{v(\theta)}} \nabla_\theta \mathcal{L}(\theta)$$

Adam

- Adam (Adapative Moment Estimation) builds on RMSProp but also uses moving average of the gradient

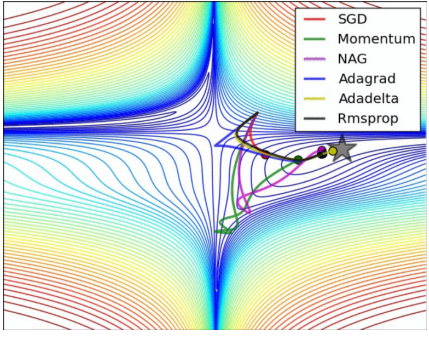
$$m(\theta) = \beta_1 m(\theta) + (1 - \beta_1) \nabla_\theta \mathcal{L}(\theta)$$
$$v(\theta) = \beta_2 v(\theta) + (1 - \beta_2) \nabla_\theta \mathcal{L}(\theta)^2$$
$$\Delta\theta = -\eta \frac{m(\theta)}{\epsilon + \sqrt{v(\theta)}}$$

- Adam = RMSProp + momentum →both of both worlds
- In practice, Adam is still the most used optimizers
- More efficient algorithms than Adam now: LARS (Layerwise Adaptive Rate Scaling), LAMB = LARS + Adam

Visualization

Comparison of modern optimizers

- These different optimizers are all variants of SGD
- SGD momentum should allow for better solution, but hyperparameters harder to find
- Adam is easier to tune



Plan

Weight initialization

- Deep learning algorithms converges iteratively
- **Initialization**: Parameters (weights of the network) need initial values
- Initialization can have an impact on:
  - convergence or not
  - convergence quality (speed, minimum)
  - generalization error
- $\rightarrow$Difficult question !
  - Initial parameters can help optimization (training)
  - but bad for generalization error !

Principle: break symmetries

- 2 identical parameters connected to same input should be initialized differently
- otherwise they will learn the same thing...

Weight initialization

Initialization to zero is **bad**→no learning... **Initialization schemes:**

- Random (uniform, gaussian)
- Example: Xavier init.

$$W_{i,j} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right)$$

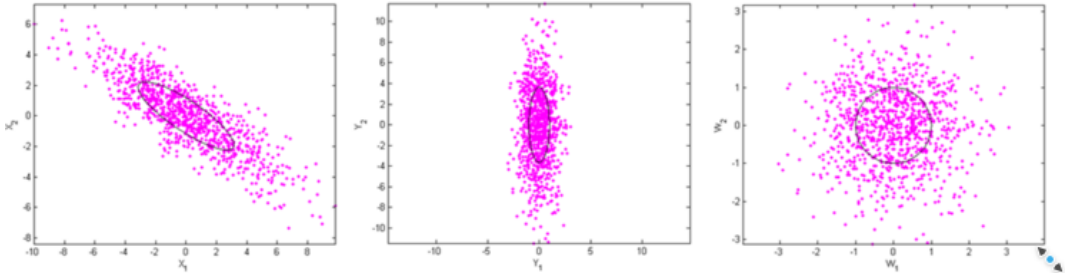
- Example: He init. n_i number of input to layer

$$W_{i,j} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_i}}\right)$$

- others: orthogonal, delta-orthogonal...

Input normalization

- Gradient descent is sensitive to strong variations in input
- Ideally, surface of loss function should have a uniform curvature in all directions $\rightarrow$sphere
- can be obtained by input normalization



Batch normalization

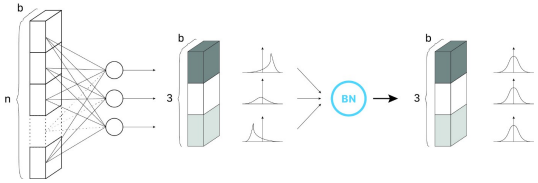
- "Adaptive reparametrization" method
- Motivation:
  - Deep NN are compositions of functions
  - parameters are iteratively updated during training
  - update done simultaneously to **all** layers
  - unexpected effects can come into play $\rightarrow$we supposed other layers constant
  - updates to other layers can add high order effects that can lead to *gradient explosion*

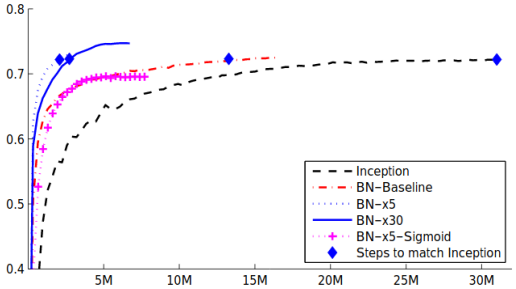
Batch normalization

Normalize distribution of each input feature in each layer across each minibatch to $\mathcal{N}(0,1)$:

$$\begin{aligned}\mu &\leftarrow \frac{1}{m}\sum_{i=1}^{m}\overline{\mathbf{x}}^i \\ \sigma^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}\left(\overline{\mathbf{x}}^i-\mu\right)^2 \\ \overline{\mathbf{x}}^i &\leftarrow \frac{\overline{\mathbf{x}}^i-\mu}{\sqrt{\sigma^2+\epsilon}}\end{aligned} \tag{1}$$

Batch normalization



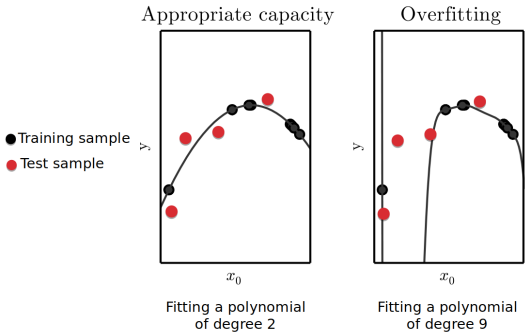


Plan

Generalization and overfitting

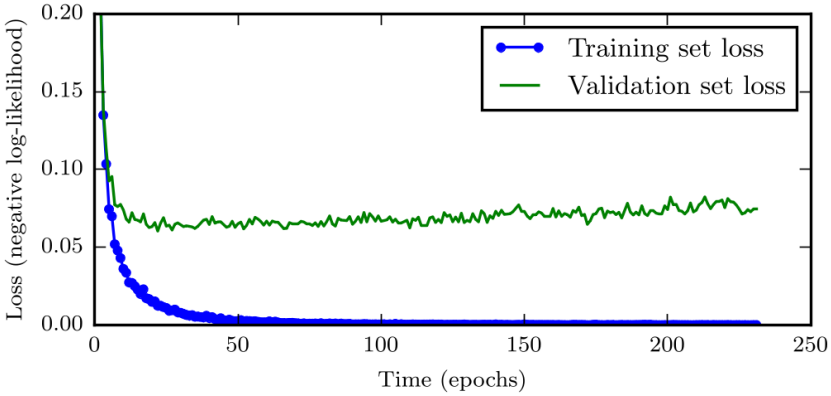
Loss function can be small on training data but large on test data: →**Overfitting**.

- Deep NN overfit: more depth = more free parameters = higher VC dimension (recall $\epsilon = \frac{\mathrm{VC}_{\mathrm{dim}}}{N}$).
- Need to put constraints on the weights to reduce VC dimension →**Regularization**



Preventing overfitting

Easy solution: Use a *validation set*. Monitor the loss on the validation set during training, stop when it starts increasing:



L2 regularization

$\mathcal{L}_2$ regularization keeps the $\mathcal{L}_2$ norm of the free parameters $\|\theta\|$ as small as possible during learning.

$$\|\theta\|^2 = w_1^2 + w_2^2 + \cdots + w_M^2$$

Each neuron will use all its inputs with small weights, instead on specializing on a small part with high weights. Two things have to be minimized at the same time: the training loss and a penalty term representing the norm of the weights:

$$\mathcal{L}(\theta) = \mathbb{E}_{\mathcal{D}}\left(\|\mathbf{t} - \mathbf{y}\|^2\right) + \lambda\|\theta\|^2$$

The regularization parameter λ controls the strength of regularization:

- if λ is small, there is only a small regularization, the weights can increase.
- if λ is high, the weights will be kept very small, but they may not minimize the training loss.

L2 regularization (Ridge)

The gradient of the new loss function is easy to find:

$$\nabla_\theta \mathcal{L}(\theta) = -2(\mathbf{t} - \mathbf{y})\nabla_\theta \mathbf{y} + 2\lambda\theta$$

The parameter updates become:

$$\Delta\theta = \eta(\mathbf{t} - \mathbf{y})\nabla_\theta \mathbf{y} - \eta\lambda\theta$$

$\mathcal{L}_2$ regularization leads to weight decay: even if there is no output error, the weight will converge to 0 . This forces the weight to constantly learn: it can not specialize on a particular example anymore (overfitting) and is forced to generalize.

L1 regularization

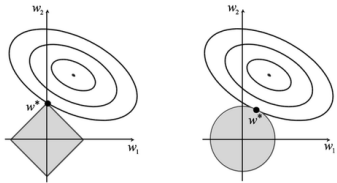
$\mathcal{L}_1$ regularization penalizes the absolute value of the weights instead of their Euclidian norm:

$$\mathcal{L}(\theta) = \mathbb{E}_{\mathcal{D}}\left[||\mathbf{t} - \mathbf{y}||^2\right] + \lambda|\theta|$$

It leads to very sparse representations: a lot of neurons will be inactive, and only a few will represent the input. Also called *Lasso* regularization.

L2 vs L1 regularization

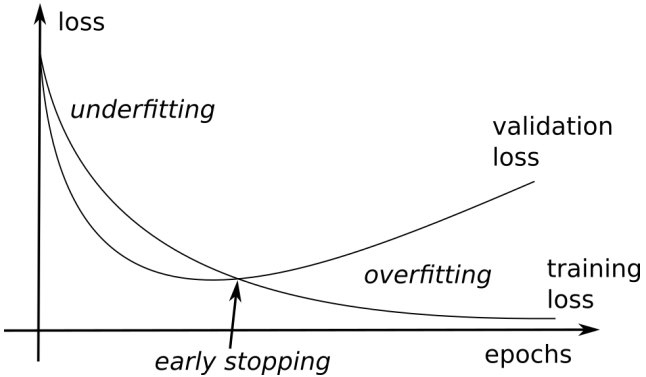
- Cost function regularized with L1 or L2
- Compromise solution = intersection between ellipsoid (loss function) and Lx ball



Early stopping

During training

- training error decreases
- validation error decreases... then increases again !

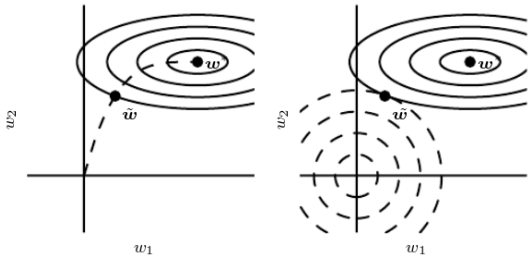


Early stopping

Early stopping

Equivalent to L2 normalization $\rightarrow$limit capacity of the model.

- L2 regulaziation:
  - small slope regions contract dimension of θ $\rightarrow$decayed to 0
  - high slope regions not regularized because they help descent
- *early stopping*:
  - parameters with high slope are learned before parameters with low slope



Dropout

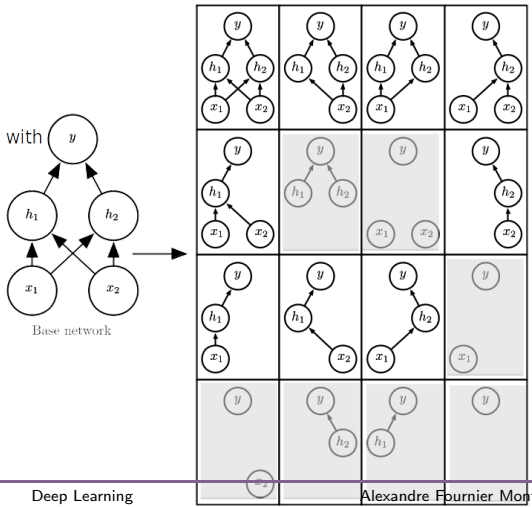
Bagging/Ensemble methods: Averaging different models, as different models will usually not make all the same errors on the test set (AdaBoost, Random Forests, etc.).

$$\tilde{o}(\mathbf{x}) = \frac{1}{N}\sum_{i}^{N} o_i(\mathbf{x}),$$

where the o_i are different models (classifiers, regressors,..), and $\tilde{o}$ the final model. DropOut is an ensemble method that does not need to build the models explicitly. Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: JMLR(2014).

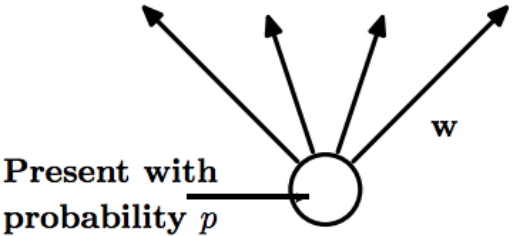
Dropout

Considers all the networks that can be formed by removing units from a network:

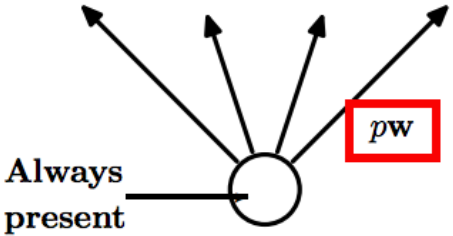


Dropout

At each optimization iteration: random binary masks on the units to consider.
The probability p to remove a unit is a hyperparameter.



(a) At training time



(b) At test time

Dropout(justification)

Exact inference:

$$\tilde{o}(\mathbf{x}) = \frac{1}{N} \sum_{i}^{N} o\left(\mathbf{x}; \mu_i\right),$$

where the o_i are different models (e.g. classifiers), $\tilde{o}$ the final model, and μ_i is the binary mask. This is however intractable. DropOut provides an approximation (that works well in practice). DropOut is exact in the case of linear classification:

$$o(\mathbf{x}) = \mathrm{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}),$$

and when using the geometric mean (instead of the arithmetic mean) to average the models):

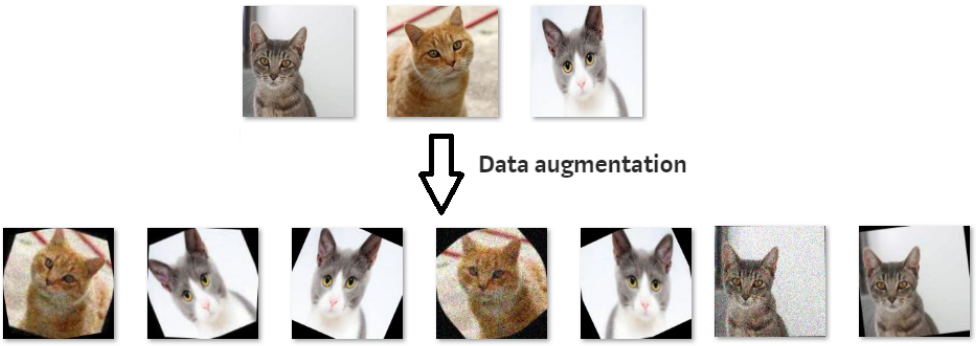
$$\tilde{o}(\mathbf{x}) = \sqrt[2^n]{\prod_{\mu \in \{0,1\}^n} o(\mathbf{x}; \mu)},$$

where:

$$o(\mathbf{x}; \mu) = \mathrm{softmax}(\mathbf{W}(\mu \odot \mathbf{x}) + \mathbf{b})$$

Data augmentation

Best way to avoid overfitting is more data (not always possible). Simple trick: **data augmentation** artificially creates new varied data by perturbating the input while keeping the label constant.



Mixup

Particular case of data augmentation.

- Creates new training samples with labels by interpolation

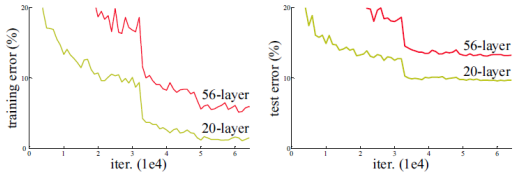
$$\tilde{x} = \lambda x_i + (1-\lambda) x_j$$
$$\tilde{y} = \lambda y_i + (1-\lambda) y_j$$

- And $\lambda \sim \mathrm{Beta}(\alpha, \alpha)$
  - $\alpha \in [0.1, 0.4]$ for classification
  - works for structured data
  - stabilizes GANs (see later course)

Plan

Vanishing gradients

Contrary to what we could think, adding more layers to a DNN does not necessarily lead to a better performance, both on the training and test set. Here is the performance of neural networks with 20 or 56 layers on CIFAR-10:



The main reason behind this is the **vanishing gradient problem**. The gradient of the loss function is repeatedly multiplied by a weight matrix W as it travels backwards in a deep network.

$$\frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} = f'\left(W^k \mathbf{h}_{k-1} + \mathbf{b}^k\right) W^k$$

Vanishing gradients

When it arrives in the first FC layer, the contribution of the weight matrices is comprised between:

$$(W_{\min})^d \quad \text{and} \quad (W_{\max})^d$$

where $W_{\max}$ (resp. $W_{\min}$) is the weight matrix with the highest (resp. lowest) norm, and d is the depth of the network.

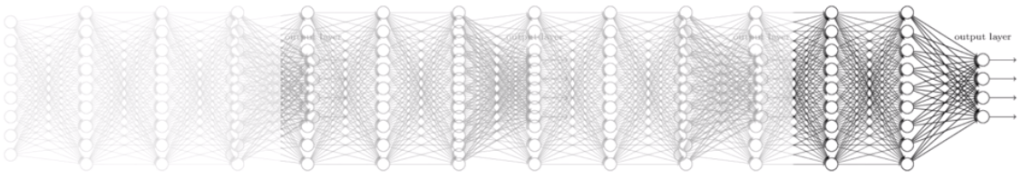
- If $|W_{\max}| < 1$, then $(W_{\max})^d$ is very small for high values of d : **the gradient vanishes**.
- If $|W_{\min}| > 1$, then $(W_{\min})^d$ is very high for high values of d : **the gradient explodes**.

Exploding gradients can be solved by **gradient clipping**, i.e. normalizing the backpropagated gradient if its norm exceeds a threshold.

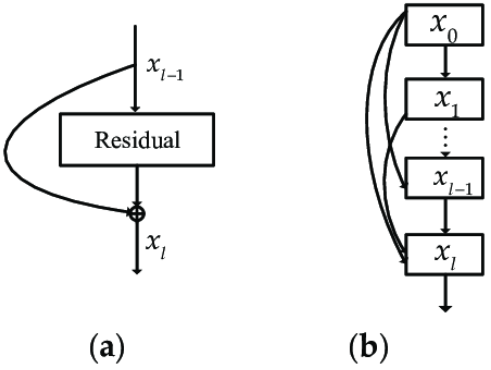
$$\left\| \frac{\partial \mathcal{L}(\theta)}{\partial W^k} \right\| \leftarrow \min \left(\left\| \frac{\partial \mathcal{L}(\theta)}{\partial W^k} \right\|, \text{ MAX\_GRAD} \right)$$

Vanishing gradients

Vanishing gradients are still the current limitation of deep networks. The solutions include: ReLU activation functions, unsupervised pre-training, batch normalization, **residual networks**...

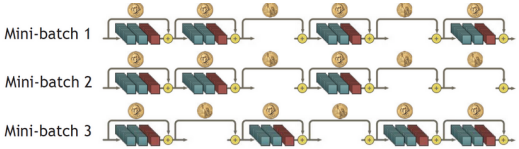


Residual networks

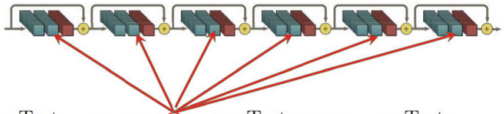


Stochastic depth

Training



Inference



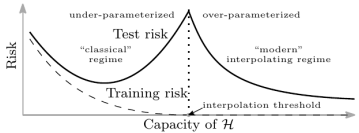
$$H_\ell^{\text{Test}} = \texttt{ReLU}(p_\ell f_\ell(H_{\ell-1}^{\text{Test}}; W_\ell) + H_{\ell-1}^{\text{Test}})$$

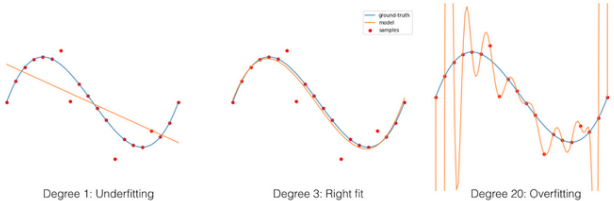
Plan

Deep double descent

- Practice of deep learning is in contradiction with "classical" statistical theory: bias-variance tradeoff seems false!?

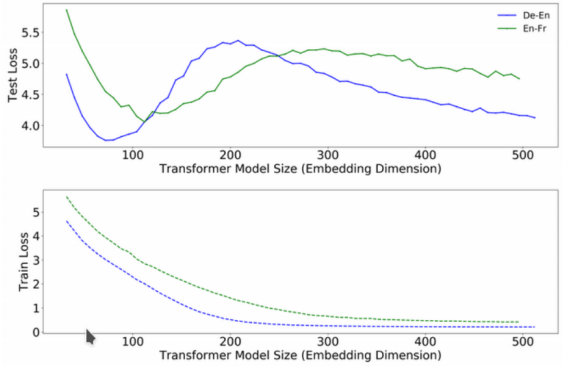
- "Big" models have good generalization capabilities
- Better generalization after training data interpolation

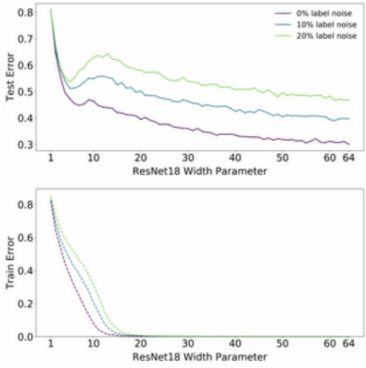




Deep double descent

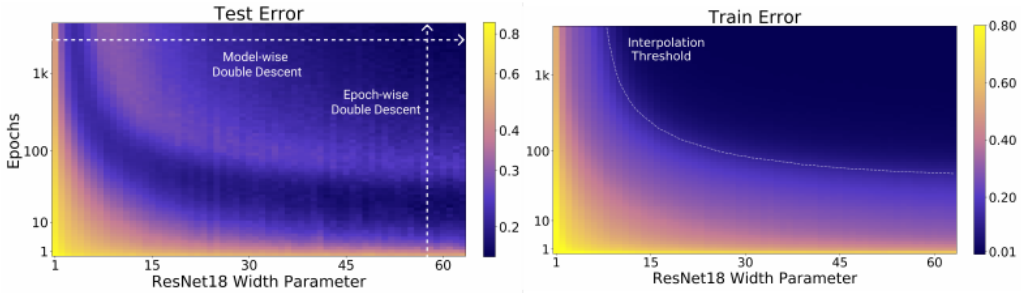
Observed on multiple architectures and tasks: ResNet-18 in image classification (CIFAR-100), Transformer (translation). Observed better **with noisy labels**.





Grokking

- Same phenomenon when we increase training time (*grokking*):
  - although needs significant capacity, shallow models don't show it



Plan

Hardware

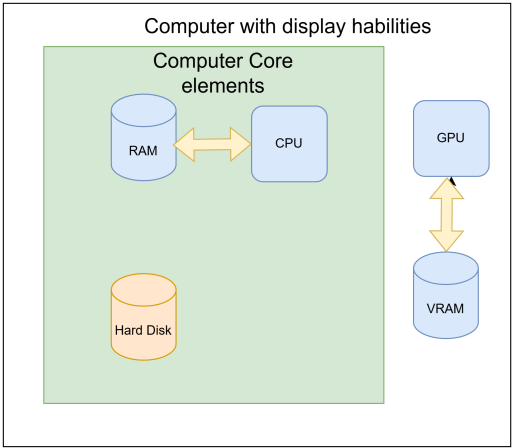


Figure: Main Hardware Components for DL

Hardware Description

- **CPU**: Performs most of the computations within a computer, executes programs, input/output operations, etc. Should be used for iterative process or parallel processes when $N_{processes} < 100$ (nb of cores is usually around 20)
- **GPU**: Process unit initially designed for 3D rendering and image processing, and is actively used for numerous parallel processes (e.g. in deep learning). Should be used for parallel processes when $N_{processes} > 100$ (Usually thousands of cores)
- **RAM** (Random Access Memory): Volatile memory used by the CPU to store any data it needs for computations
- **VRAM** (Video Random Access Memory): Same as RAM for CPU but for GPU

Hardware Transfer

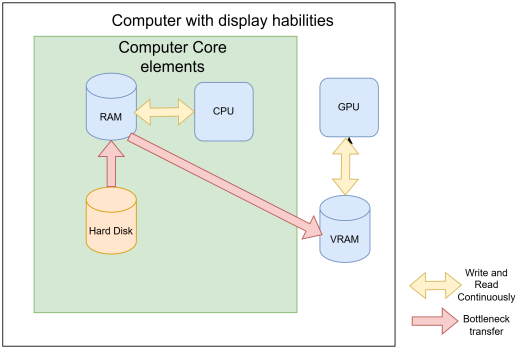


Figure: Loading Data

Listing: Training Loop

```
for epoch in range(10):  # 10 epochs
    for batch in dataloader:
        inputs, labels = batch
        optimizer.zero_grad()
        outputs = model(inputs.to("cuda"))
        loss = criterion(outputs, labels.cuda())
        loss.backward()
        optimizer.step()
```

Listing: Variables

```
dataloader = DataLoader(dataset, batch_size=16, shuffle=True, num
model = SimpleModel().cuda()
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Listing: Model

```
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.layer = nn.Linear(10, 1)

    def forward(self, x):
        return torch.sigmoid(self.layer(x))
```

Listing: Dataset

```
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```