Introduction to Deep Learning

Graduate School Orléans Numérique

Lecture 2: how a network learns

Daniel Förster
UFR Sciences et Techniques
Université d'Orléans

Agenda

- Recap: what training has to do
- Gradient descent: learning rate, momentum, Adam, mini-batches
- Backpropagation: the chain rule, layer by layer
- Vanishing gradients and weight initialisation
- Generalisation: overfitting, early stopping, data augmentation, regularisation



`daniel-forster.fr/dl`

slides, problem sheets, notebooks, project ideas

Recap: what training has to do

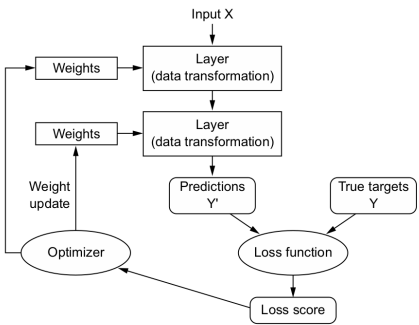


Image source: B. Pesquet, ML handbook

Forward pass: layer by layer, $y_j = \phi\left(\sum_i x_i w_{ij} + b_j\right)$, up to the prediction $\hat{y}$ and the loss $E(\hat{y}, y)$ ($\mathcal{L}$ in lecture 1)

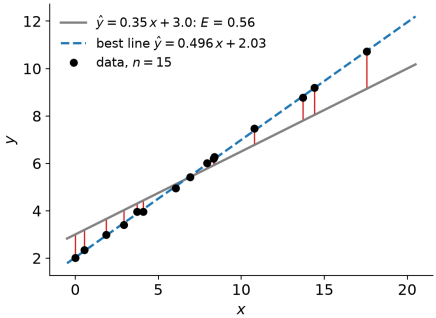
Practical 1: `Dense(16)` on MNIST, 12 730 parameters, about 95 % accuracy after a few epochs. `model.fit` found the parameters. **How?**

The perceptron rule does not extend to hidden layers: the derivative of a step function is zero, so there is no signal to pass back.

Today:

- walk downhill on the loss: **gradient descent**
- get the slope for every weight: **backpropagation**
- make it work on new data: **regularisation**

The smallest network: linear regression



One input, one neuron, linear activation: $\hat{y} = wx + b$, with weight w (slope) and bias b (intercept)

Loss: mean squared error over the n points (x_j, y_j)

$$E(w, b) = \frac{1}{n} \sum_{j=1}^{n} (y_j - \hat{y}_j)^2, \qquad \hat{y}_j = wx_j + b$$

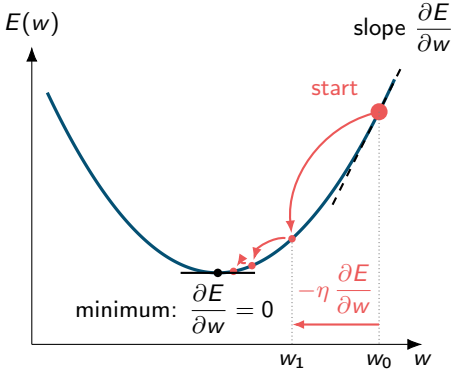
Gradient (chain rule on the square):

$$\frac{\partial E}{\partial w} = -\frac{2}{n} \sum_{j=1}^{n} (y_j - \hat{y}_j)\, x_j$$

$$\frac{\partial E}{\partial b} = -\frac{2}{n} \sum_{j=1}^{n} (y_j - \hat{y}_j)$$

A network has no closed-form solution (here: least squares): we need a method that only uses the gradient.

Gradient descent



All parameters in one vector p (here $\mathrm{p} = (w, b)$).
Repeat:

$$\mathrm{p} \leftarrow \mathrm{p} - \eta \nabla E(\mathrm{p}), \qquad \nabla E = \left(\frac{\partial E}{\partial w}, \frac{\partial E}{\partial b} \right)$$

- ∇E points uphill: step the other way
- step $\propto$ slope: large far from the minimum, small near it, zero where $\nabla E = 0$
- all parameters are updated at the same time
- $\eta > 0$: the **learning rate**, the most important hyperparameter

Algorithm: random start; loss and gradient; update; repeat until the loss stops decreasing.

Why does it work?

Taylor expansion around the current point p, for a small step $\boldsymbol{\delta}$:

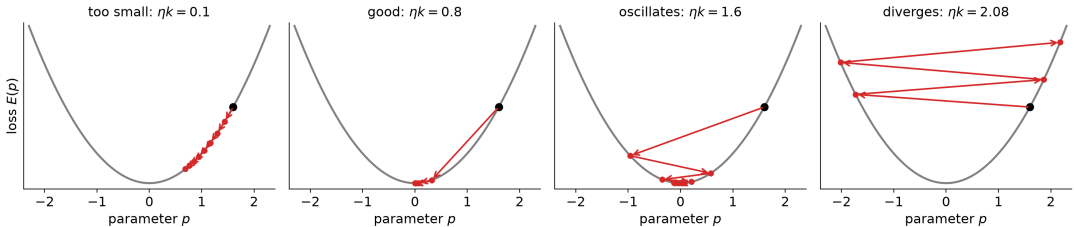
$$E(\mathrm{p}+\boldsymbol{\delta}) = E(\mathrm{p}) + \nabla E(\mathrm{p}) \cdot \boldsymbol{\delta} + \tfrac{1}{2}\,\boldsymbol{\delta}^T H \boldsymbol{\delta} + \ldots \qquad H: \text{ matrix of second derivatives (curvature)}$$

Gradient descent chooses $\boldsymbol{\delta} = -\eta \nabla E(\mathrm{p})$:

$$E\left(\mathrm{p} - \eta \nabla E\right) \approx E(\mathrm{p}) - \eta \left\|\nabla E(\mathrm{p})\right\|^2 + \mathcal{O}(\eta^2) \;<\; E(\mathrm{p}) \qquad \text{for small enough } \eta$$

- Every step lowers the loss, **if** η is small enough: the curvature term $\frac{1}{2}\boldsymbol{\delta}^T H \boldsymbol{\delta}$ sets how small (next slide)
- The method is **local**: it stops at any point where $\nabla E = 0$: a minimum, a saddle point, a flat plateau
- Global minimum guaranteed only for a **convex** loss: linear regression yes, neural networks no
- In practice the minima found by descent in networks are good enough; in millions of dimensions, most points with $\nabla E = 0$ are saddle points, not bad local minima

The learning rate

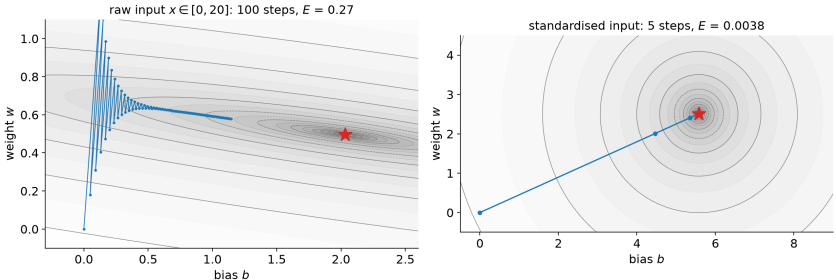


Exactly solvable case: a parabola $E = \frac{1}{2}kp^2$ with curvature k. One step gives

$$p \leftarrow p - \eta k p = (1 - \eta k)\, p \qquad \Rightarrow \qquad \text{converges if and only if } 0 < \eta k < 2$$

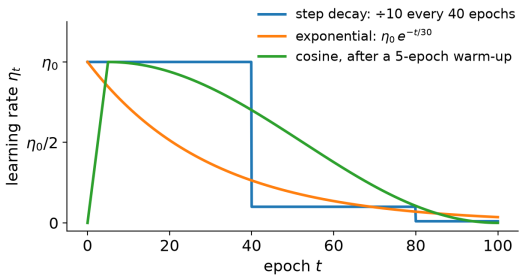
Fastest for $\eta k = 1$ (one step); oscillates for $1 < \eta k < 2$; diverges for $\eta k > 2$. In practice: try $\eta = 10^{-1}, 10^{-2}, 10^{-3}, \ldots$ and watch the training loss. A loss that grows or turns into `nan`: η is too large.

Two directions, one learning rate



- **Left**, raw input $x \in [0, 20]$: the curvatures along the two axes of the valley differ by a factor 230. η must suit the steep direction ($\eta k_{\max} < 2$): zig-zag, then a slow crawl along the flat one (the minimum is $E = 0.0038$)
- **Right**, standardised input $(x - \bar{x})/\sigma_x$: a round bowl, the minimum in 5 steps
- $\Rightarrow$ **Why we normalise the inputs** (lecture 1). Practical 1: MNIST with a white background (1 instead of 0) trained much worse, the same effect in 784 dimensions

Learning-rate schedules



Large steps at the start (fast progress), small steps at the end (settle into the minimum instead of bouncing around it).

- **Step decay:** divide η by a factor every few epochs
- **Exponential:** $\eta_t = \eta_0\, e^{-\lambda t}$
- **Cosine:** half a cosine down to zero
- **Warm-up:** start from $\eta = 0$ for the first steps (large models, transformers)
- **Reduce on plateau:** divide η when the validation loss stalls

Keras:
`keras.optimizers.schedules.CosineDecay`, `ExponentialDecay`, callback `ReduceLROnPlateau`

Momentum

Average the gradient over the last steps:

$$\mathsf{m} \leftarrow \beta\,\mathsf{m} + (1-\beta)\,\nabla E(\mathsf{p})$$
$$\mathsf{p} \leftarrow \mathsf{p} - \eta\,\mathsf{m}$$

$\beta \approx 0.9$: an average over about $1/(1-\beta) = 10$ steps.

- Directions where the gradient keeps its sign **add up**: fast progress along the valley
- Directions where it flips sign (the zig-zag) **cancel**
- Much larger learning rates stay stable: on our example up to $\eta \approx 0.24$ instead of 0.013

Physics picture (Polyak, 1964): a heavy ball rolling down the loss surface with friction

- $-\nabla E$: the force
- m: the velocity
- $1-\beta$: the friction

Without momentum, the walker takes each local slope at face value; with momentum, it builds up speed where the slope is consistent.

One step size per direction: RMSprop and Adam

RMSprop (Hinton, 2012): divide each direction by its typical gradient size (element by element)

$$\mathsf{v} \leftarrow \beta_2\,\mathsf{v} + (1-\beta_2)\,(\nabla E)^2, \qquad \mathsf{p} \leftarrow \mathsf{p} - \eta\,\frac{\nabla E}{\sqrt{\mathsf{v}}+\epsilon}$$

Steep directions get small steps, flat directions large steps; ϵ avoids a division by zero.

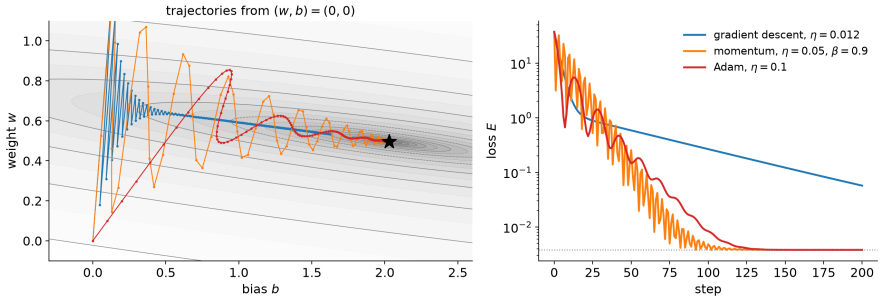
Adam, adaptive moment estimation (Kingma & Ba, 2014) = momentum + RMSprop + a correction:

$$\mathsf{m} \leftarrow \beta_1\mathsf{m} + (1-\beta_1)\nabla E, \qquad \mathsf{v} \leftarrow \beta_2\mathsf{v} + (1-\beta_2)(\nabla E)^2$$

$$\hat{\mathsf{m}} = \frac{\mathsf{m}}{1-\beta_1^t}, \qquad \hat{\mathsf{v}} = \frac{\mathsf{v}}{1-\beta_2^t}, \qquad \mathsf{p} \leftarrow \mathsf{p} - \eta\,\frac{\hat{\mathsf{m}}}{\sqrt{\hat{\mathsf{v}}}+\epsilon} \qquad (t = 1, 2, \ldots \text{ counts the steps})$$

- m, v start at zero: at $t = 1$, $\mathsf{m} = 0.1\,\nabla E$, ten times too small; dividing by $1-\beta_1^t$ corrects this
- Every step has a size of about η in each direction, whatever the scale of the gradient: one default works for most problems
- Keras defaults: $\eta = 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$: the `optimizer="adam"` of practical 1

Comparison on the regression example



- **Gradient descent**, near the largest stable η: zig-zags, then crawls; after 200 steps still 15 times above the minimum
- **Momentum**: overshoots and oscillates, but races along the valley
- **Adam**: first steps of size η in both directions, then follows the valley floor

Mini-batches and epochs

The loss is an average over the n training examples, and so is its gradient:

$$E = \frac{1}{n}\sum_{j=1}^{n} E_j \qquad \Rightarrow \qquad \nabla E = \frac{1}{n}\sum_{j=1}^{n} \nabla E_j$$

- **(Full-)batch gradient descent:** all n examples for every step. Exact, but slow
- **Stochastic gradient descent (SGD):** one random example per step. Cheap, very noisy
- **Mini-batch gradient descent:** B random examples per step (typically 32–256): a cheap, noisy estimate of ∇E, noise $\propto 1/\sqrt{B}$. B examples at once = one matrix product, what GPUs are built for
- **Epoch:** one pass over the training set = $\lceil n/B \rceil$ steps

Practical 1: 49 000 training images, batch size 32 (Keras default) $\rightarrow \lceil 49\,000/32 \rceil = 1\,532$ steps per epoch: the counter `1532/1532` of `model.fit`.
The noise also helps: it shakes the parameters out of saddle points and flat regions.

Backpropagation: the problem

Gradient descent on every weight and bias of every layer:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}, \qquad b_j \leftarrow b_j - \eta \frac{\partial E}{\partial b_j}$$

Naive approach, finite differences: change one parameter a little and see what the loss does,

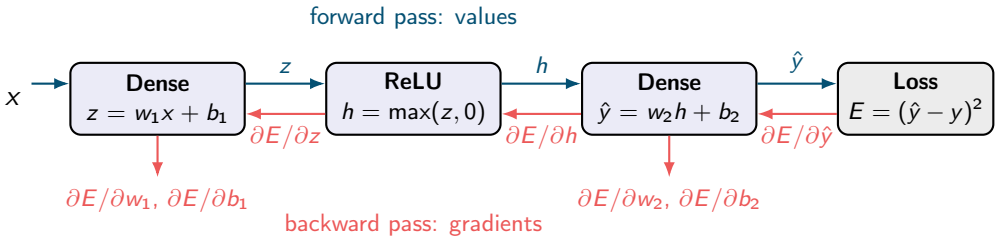
$$\frac{\partial E}{\partial p} \approx \frac{E(p+\varepsilon) - E(p)}{\varepsilon}$$

One extra forward pass **per parameter**: 12 731 forward passes for one step of the practical 1 network; for GPT-3 (175×10^9 parameters), hopeless.

Backpropagation: all derivatives at once, for the cost of about two to three forward passes, whatever the number of parameters. Idea: the chain rule, applied from the loss back to the input, reusing each intermediate result.

Reverse-mode differentiation: Linnainmaa (1970); for neural networks: Werbos (1974), popularised by Rumelhart, Hinton & Williams, *Nature* (1986).

The chain rule on a computational graph



The loss depends on w_1 only through z, h and $\hat{y}$: multiply the local derivatives,

$$\frac{\partial E}{\partial w_1} = \underbrace{\frac{\partial E}{\partial \hat{y}}}_{2(\hat{y}-y)} \cdot \underbrace{\frac{\partial \hat{y}}{\partial h}}_{w_2} \cdot \underbrace{\frac{\partial h}{\partial z}}_{1 \text{ if } z>0,\ 0 \text{ else}} \cdot \underbrace{\frac{\partial z}{\partial w_1}}_{x}$$

Each box needs only its own input (stored during the forward pass) and the gradient arriving from its right. The product is built from right to left, and each partial product is reused by the next box.

Your turn: one training step by hand

Same network as on the previous slide:
input $x = 1$, target $y = 1$,
$w_1 = 2$, $b_1 = -1$, $w_2 = 3$, $b_2 = 0$.

Forward:

$$z = w_1 x + b_1 = 1, \qquad h = \max(z, 0) = 1$$
$$\hat{y} = w_2 h + b_2 = 3, \qquad E = (\hat{y} - y)^2 = 4$$

Backward: compute $\partial E/\partial w_2$, $\partial E/\partial w_1$ and $\partial E/\partial b_1$.

Update with $\eta = 0.01$: what is the new loss?

Your turn: one training step by hand

Same network as on the previous slide:
input $x = 1$, target $y = 1$,
$w_1 = 2$, $b_1 = -1$, $w_2 = 3$, $b_2 = 0$.

Forward:

$$z = w_1 x + b_1 = 1, \qquad h = \max(z, 0) = 1$$
$$\hat{y} = w_2 h + b_2 = 3, \qquad E = (\hat{y} - y)^2 = 4$$

Backward: compute $\partial E/\partial w_2$, $\partial E/\partial w_1$ and $\partial E/\partial b_1$.

Update with $\eta = 0.01$: what is the new loss?

$$\partial E/\partial \hat{y} = 2(\hat{y} - y) = 4$$
$$\partial E/\partial w_2 = 4 \cdot h = 4, \quad \partial E/\partial b_2 = 4$$
$$\partial E/\partial h = 4 \cdot w_2 = 12$$
$$\partial E/\partial z = 12 \cdot 1 = 12 \quad (z > 0)$$
$$\partial E/\partial w_1 = 12 \cdot x = 12, \quad \partial E/\partial b_1 = 12$$

New: $w_1 = 1.88$, $b_1 = -1.12$, $w_2 = 2.96$, $b_2 = -0.04$
$\Rightarrow \hat{y} = 2.21$, $E = 1.46$ (was 4)
Check by finite difference: $w_1 = 2.001$ gives $E = 4.012$, slope ≈ 12

Dense layer: the three gradients

One example, $y_j = \sum_i x_i w_{ij} + b_j$.
Given $\partial E/\partial y_j$ from the layer above:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\, x_i$$
$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_j}$$
$$\frac{\partial E}{\partial x_i} = \sum_j \frac{\partial E}{\partial y_j}\, w_{ij}$$

The sum: x_i feeds every output y_j, all contributions add up. $\partial E/\partial x_i$ goes on to the layer below.

A mini-batch (as in NumPy and Keras): the B examples are the rows of X,

$$Y = XW + \mathrm{b}, \qquad X : B \times n_{\text{in}}, \quad W : n_{\text{in}} \times n_{\text{out}}$$

With $G = \partial E/\partial Y$, of size $B \times n_{\text{out}}$:

$$\frac{\partial E}{\partial W} = X^T G, \qquad \frac{\partial E}{\partial \mathrm{b}} = \sum_{\text{batch}} G, \qquad \frac{\partial E}{\partial X} = G\, W^T$$

- The shapes fix the formula: $X^T G$ is $n_{\text{in}} \times n_{\text{out}}$, like W
- The backward pass reuses W, transposed, and the input X of the forward pass: store it
- The gradients of the batch examples add up

Activation and output layers

Activation layer, $y_i = f(x_i)$, element by element, no parameters:

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_i} f'(x_i)$$

- **ReLU:** $f'(x) = 1$ if $x > 0$, else 0: the gradient passes unchanged or is blocked
- **Sigmoid:** $\sigma' = \sigma(1 - \sigma) \leq \frac{1}{4}$: the gradient shrinks at every layer
- **tanh:** $f'(x) = 1 - \tanh^2 x \leq 1$

Where the backward pass starts

MSE, linear output:

$$E = \frac{1}{n} \sum_j (\hat{y}_j - y_j)^2 \;\Rightarrow\; \frac{\partial E}{\partial \hat{y}_j} = \frac{2}{n}(\hat{y}_j - y_j)$$

Softmax + cross-entropy, one example of class c, softmax inputs ("logits") z_k:

$$E = -\log \hat{y}_c = -z_c + \log \sum_k e^{z_k} \;\Rightarrow\; \frac{\partial E}{\partial z_k} = \hat{y}_k - y_k$$

Prediction minus target, in both cases. The perceptron rule of lecture 1, $w \leftarrow w + \eta\,(y - \hat{y})\,x$, has the same form.

In code: a layer is a class

```
class dense():
    def __init__(self, n_in, n_out):
        std = np.sqrt(2 / n_in) # He initialisation
        self.weights = std * np.random.randn(n_in, n_out)
        self.biases = np.zeros((1, n_out))

    def forward(self, data_in):
        self.input = data_in # kept for backprop
        return data_in @ self.weights + self.biases

    def backprop(self, error_out, learning_rate):
        error_in = error_out @ self.weights.T # dE/dX
        grad_weights = self.input.T @ error_out # dE/dW
        grad_biases = error_out.sum(axis=0) # dE/db
        self.weights -= learning_rate * grad_weights
        self.biases -= learning_rate * grad_biases
        return error_in
```

Every layer has the same two methods:

- `forward`: $X \to Y$, stores what the backward pass will need
- `backprop`: receives $\partial E/\partial Y$, updates its own parameters, returns $\partial E/\partial X$

A layer without parameters (ReLU, next slide) only passes the gradient on. A network is a list of layers.

In code: ReLU and the training loop

```
class relu():
    def forward(self, data_in):
        self.input = data_in
        return np.maximum(data_in, 0)

    def backprop(self, error_out, learning_rate):
        return error_out * (self.input > 0)
```

```
network = [dense(784, 64), relu(), dense(64, 10)]
for epoch in range(5):
    for x, y in batches(x_train, y_train, 32):
        out = x
        for layer in network: # forward
            out = layer.forward(out)
        # softmax + cross-entropy: dE/dz
        error = (softmax(out) - one_hot(y)) / len(x)
        for layer in reversed(network): # backward
            error = layer.backprop(error, 0.1)
```

- The loss enters only once: through the first error, *prediction minus target*
- Division by the batch size: the loss is the batch average
- **MNIST**, 5 epochs of 0.6 s each on a laptop: **96–97 %** validation accuracy, without TensorFlow

→ Demo notebook
→ **Practical 2** builds on these classes

Automatic differentiation: what `model.fit` does

- TensorFlow and PyTorch **record** the forward pass (the computational graph) and apply the chain rule backwards **automatically**
- You write only the forward pass: any combination of differentiable operations works
- `model.fit` = this loop over mini-batches and epochs, plus metrics, validation, callbacks
- **Practical 2:** backpropagation by hand, once. After that, `model.fit` is no longer magic

Vanishing and exploding gradients

Through L layers, the backward signal is a **product** of L factors $f'(z)\,w$:

- factors < 1: the gradient **vanishes**, the first layers do not learn. Sigmoid: $f' \leq \frac{1}{4}$, and $(\frac{1}{4})^{10} \approx 10^{-6}$
- factors > 1: the gradient **explodes**, training diverges

Identified by Hochreiter (1991): a main reason why deep networks were hard to train before about 2010.

Remedies: ReLU ($f' = 1$ on the active side), weight initialisation (next slide), normalisation layers, shortcut connections (lecture 3)

Weight initialisation

- **All zeros:** every neuron of a layer computes the same output and receives the same gradient: they stay identical forever. Random values are needed to **break the symmetry**
- **How large?** A neuron sums n_{in} inputs, $z = \sum_{i=1}^{n_{\text{in}}} w_i x_i$. For independent, zero-mean weights:

$$\text{Var}(z) = n_{\text{in}}\ \text{Var}(w)\ \langle x^2 \rangle$$

To keep the signal the same size from layer to layer: $\text{Var}(w) \approx 1/n_{\text{in}}$

	$\text{Var}(w)$	for
Glorot / **Xavier** (2010)	$2/(n_{\text{in}} + n_{\text{out}})$, balances forward and backward	sigmoid, tanh
He (2015)	$2/n_{\text{in}}$, ReLU zeroes half of its inputs	ReLU

Biases start at zero. Keras `Dense` default: Glorot (uniform); for deep ReLU networks use `kernel_initializer="he_normal"`.

Under- and overfitting

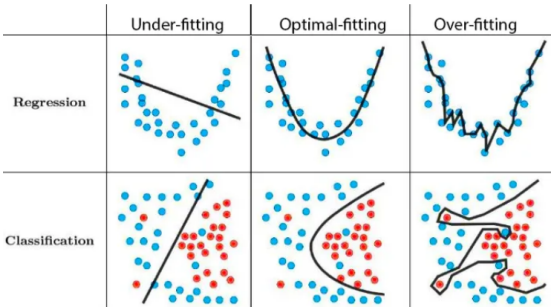


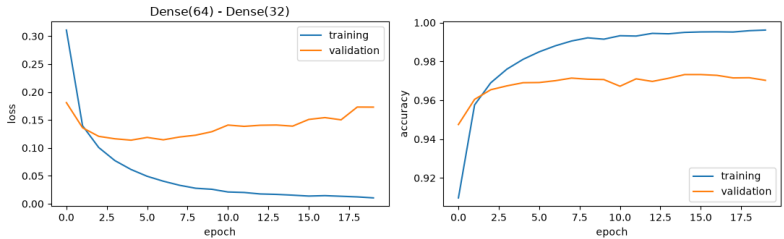
Image source: ExpoundAI

A low training loss is not the goal: we want a low loss on **new** data.

	training loss	validation loss
underfitting	high	high
good fit	low	low, close
overfitting	low, falling	rising

- **Underfitting:** model too simple, or not trained long enough
- **Overfitting:** the model memorises the training set, noise included
- Overfitting grows with the number of parameters and shrinks with the amount of data

You have seen it: practical 1, and early stopping



`Dense(64)` – `Dense(32)` on MNIST: training accuracy near 100 %, validation stuck near 97 %, validation loss lowest around epoch 4, then rising.

Early stopping: watch the validation loss, stop when it has not improved for a few epochs ("patience"), keep the best weights. The simplest and most used regulariser.

```
stop = keras.callbacks.EarlyStopping(monitor="val_loss", patience=5, restore_best_weights=True)
model.fit(x_train, y_train, epochs=100, validation_data=(x_val, y_val), callbacks=[stop])
```

Data augmentation

Enlarge the training set with transformed copies of the examples; the label does not change.

- **Against overfitting:** the network cannot memorise exact pixels
- **Simulates real variation:** position, orientation, lighting, noise
- **Compensates for small datasets**

Only transformations that leave the answer unchanged: a flipped cat is still a cat, a rotated 6 is a 9.

Keras layers `RandomFlip`, `RandomRotation`, `RandomTranslation`, `RandomZoom`: active during training only

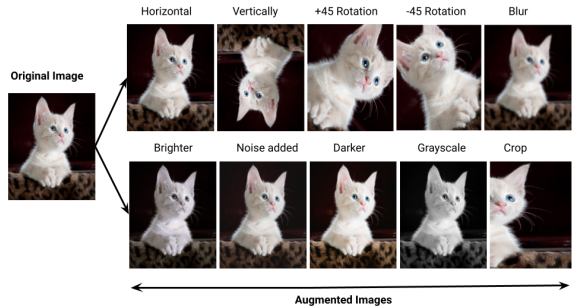


Image source: Medium

Regularisation: weight penalties and dropout

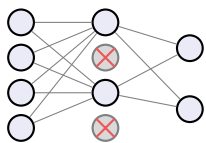
Weight penalties: a cost for large weights

L_2: $E' = E + \lambda \sum_i w_i^2$ $\quad$ L_1: $E' = E + \lambda \sum_i |w_i|$

- L_2 adds $-2\eta\lambda w$ to each step: the weights shrink by a factor $(1 - 2\eta\lambda)$ per step, "weight decay"
- L_1 pulls with a constant force: many weights become exactly zero (a sparse model)

Keras: `Dense(..., kernel_regularizer=keras.regularizers.l2(1e-4))`

Dropout (Hinton et al., 2012): at each training step, every unit is set to zero with probability p (0.2–0.5), the others are scaled by $1/(1-p)$



- No unit can rely on a particular other unit
- An ensemble of thinned networks sharing their weights
- At test time all units are on

`layers.Dropout(0.3)` after a layer

Batch normalisation

Standardising the inputs made the loss surface round. **Batch normalisation** (Ioffe & Szegedy, 2015) does the same for the inputs of every layer, for each unit over the mini-batch:

$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} h_i, \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(h_i - \mu_B)^2, \qquad \hat{h}_i = \frac{h_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \qquad h_i^{\mathrm{BN}} = \gamma\,\hat{h}_i + \beta$$

- γ and β are learned by backpropagation: the network can undo the normalisation where it does not help
- **Training:** statistics of the current mini-batch. **Inference:** running averages collected during training, e.g. $\mu \leftarrow 0.99\,\mu + 0.01\,\mu_B$ (Keras default), so that a single image can be processed
- **Effects:** more stable training, larger learning rates, a slight regularisation (each example is coupled to the others in its batch)
- Why it works so well is still debated; the recipe is standard in convolutional networks (lecture 3)

Keras: `layers.BatchNormalization()`, usually between a `Dense` or convolution layer and its activation

Hyperparameters and the validation set

Hyperparameters: everything that is not a weight or a bias, chosen by you

- number of layers and units, activation functions
- optimiser, learning rate and schedule, batch size, number of epochs
- initialisation, dropout rate, penalty λ, data augmentation

How to tune them:

- compare on the **validation** set; the **test** set is used once, at the very end
- start from a working baseline, change one thing at a time; the learning rate first
- random search beats a regular grid when only a few hyperparameters matter (Bergstra & Bengio, 2012)
- **ensembling:** average the predictions of several trained models, a few percent for a few times the cost

Every choice made by looking at validation results leaks a little of the validation set into the model: this is why the test set stays untouched.

Summary and outlook

Training is	gradient descent on the loss: $\mathrm{p} \leftarrow \mathrm{p} - \eta \nabla E(\mathrm{p})$
The learning rate	must satisfy $\eta k < 2$ in the steepest direction; normalised inputs help
Adam	momentum + one step size per direction + bias correction; the default
Mini-batches	noisy, cheap gradients; one epoch $= \lceil n/B \rceil$ steps
Backpropagation	the chain rule, from the loss back to the input, layer by layer
Deep networks	gradients vanish or explode: ReLU, He/Glorot initialisation, normalisation
Generalisation	watch the validation loss: early stopping, augmentation, L_2, dropout, batch norm

Next session, practical 2: gradient descent by hand on a function and on a linear regression, then a small network built from your own `dense` and `relu` classes.

Then: multiple-choice test 1 on lectures 1 and 2.

Lecture 3: convolutional networks, the architecture behind image recognition.