Multilayer perceptron

Victor Kitov

v.v.kitov@yandex.ru

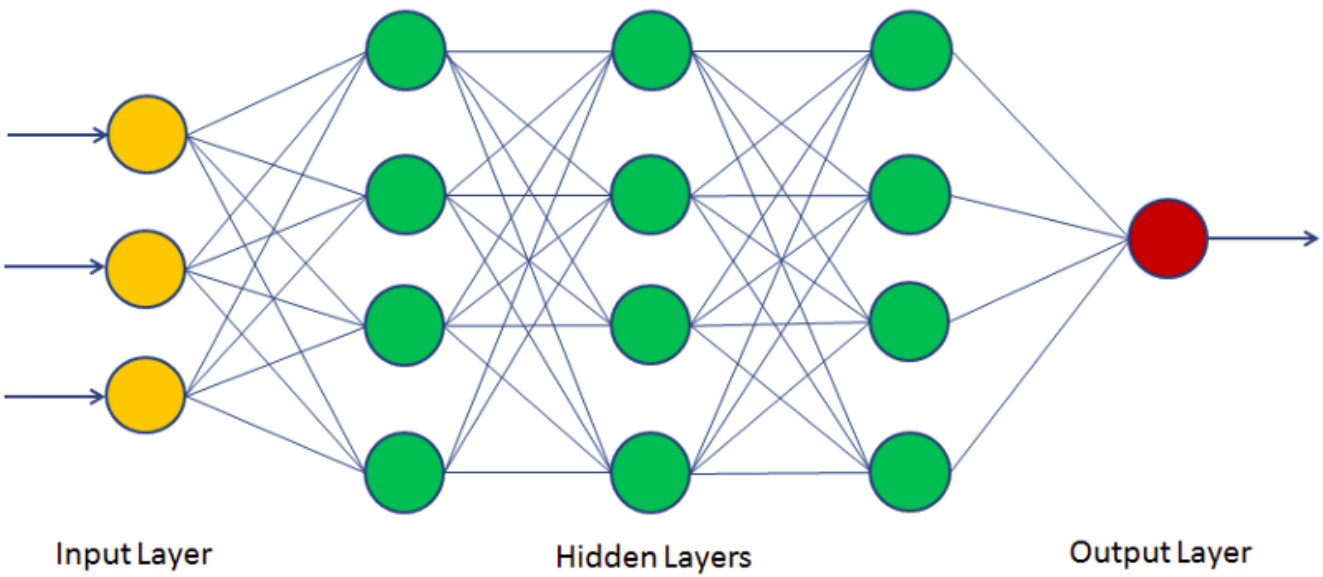


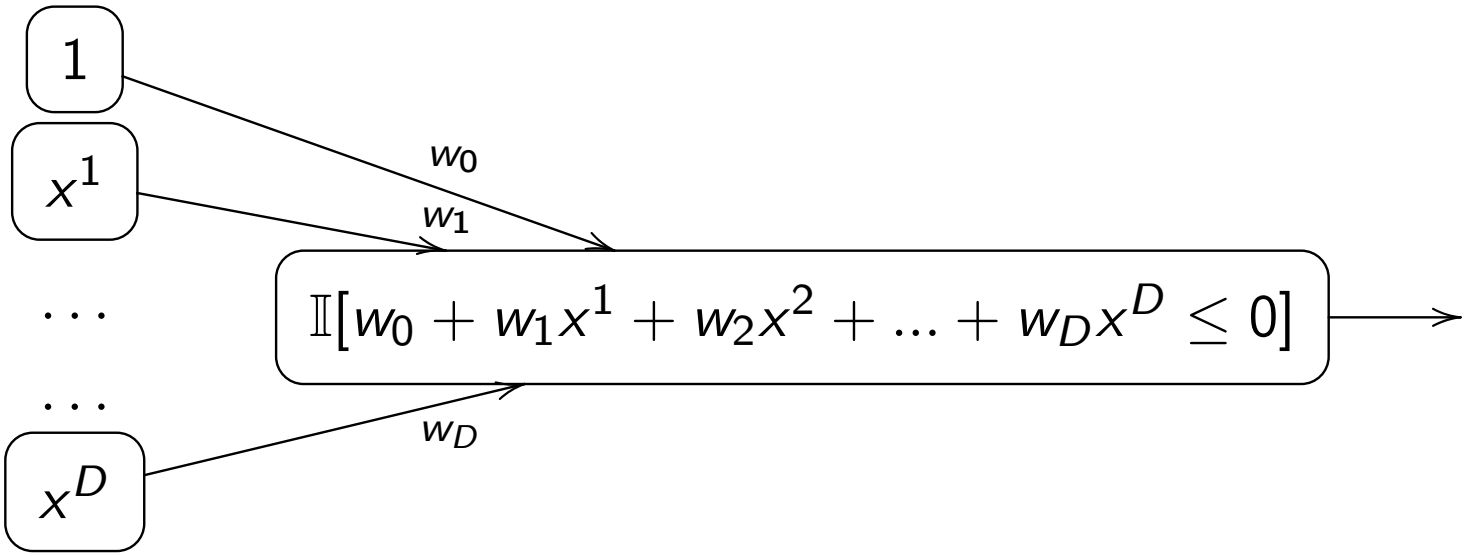
Table of Contents

History

- Neural networks originally appeared as an attempt to model human brain

- Human brain consists of multiple interconnected neuron cells
  - cerebral cortex (the largest part) is estimated to contain 15–33 billion neurons
  - communication is performed by sending electrical and electro-chemical signals
  - signals are transmitted through axons - long thin parts of neurons.

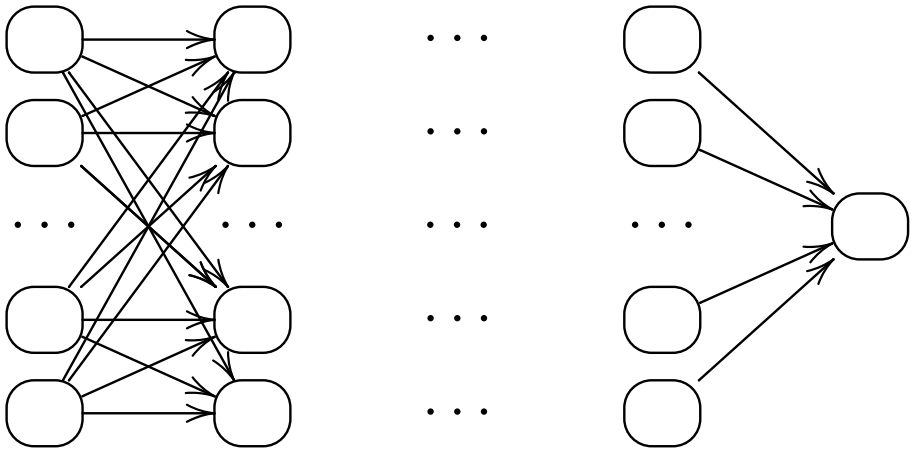
Simple model of a neuron



- Neuron get's activated in the half-space, defined by $w_0 + w_1 x^1 + w_2 x^2 + ... + w_D x^D \leq 0$.
- Each node is called a neuron
- Each edge is associated a weight
- w_0 stands for bias

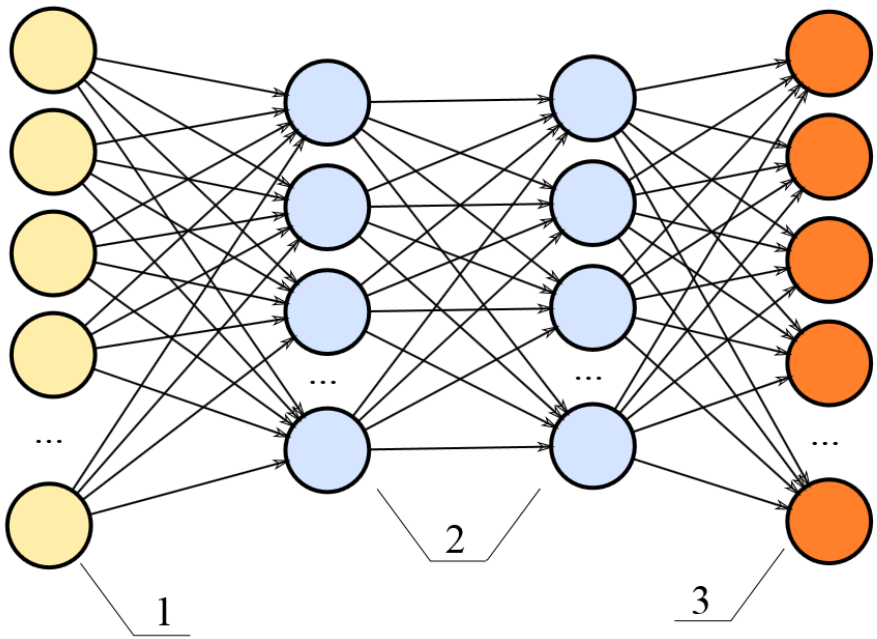
Multilayer perceptron architecture

- Hierarchically nested set of neurons.
- Each node has its own weights.



This is structure of **multilayer perceptron** - acyclic directed graph.

Layers



- Structure of neural network:
  - 1-input layer
  - 2-hidden layers
  - 3-output layer

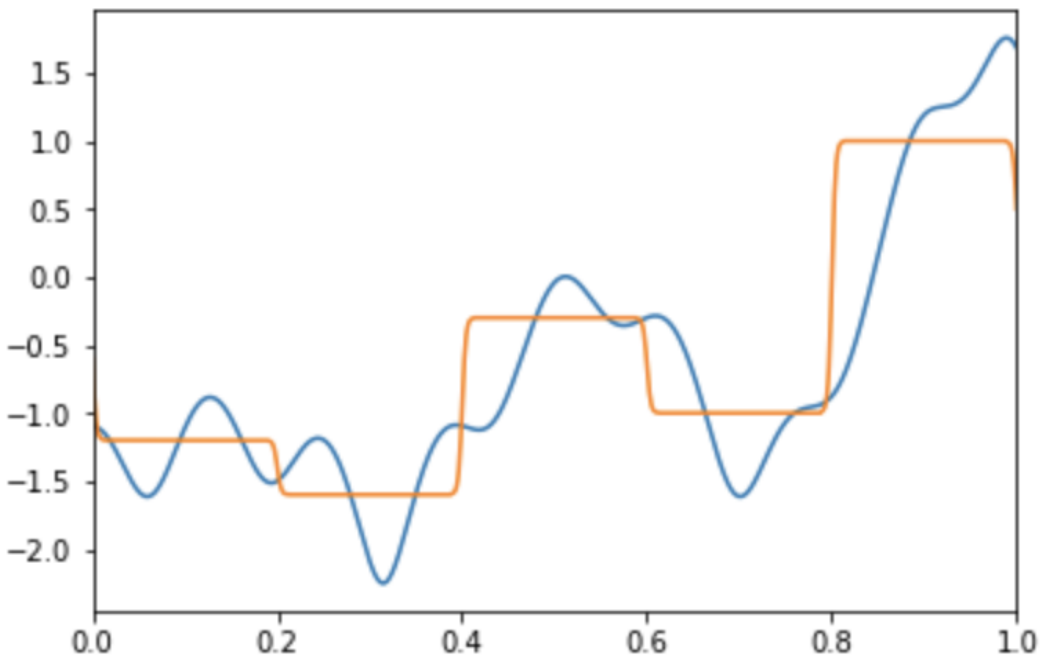
Table of Contents

Regression

- 1-dimensional regression:

$$f(x) = \sum_i f(b_i)\mathbb{I}[x \in [a_i, b_i]] = \sum_i f(b_i)\left(\mathbb{I}[x \le b_i] - \mathbb{I}[x \le a_i]\right)$$

$$= \sum_i f(b_i)\mathbb{I}[x \le b_i] - \sum_i f(b_i)\mathbb{I}[x \le a_i] \quad \text{2 layer perceptron}$$



Regression

- AND/OR function for $x_1, x_2 \in \{0, 1\}$ made with 1 layer perceptron[1]:

$$\text{AND function } \mathbb{I}[x_1 + x_2 \geq 2] = \mathbb{I}[-x_1 - x_2 \leq -2]$$
$$\text{OR function } \mathbb{I}[x_1 + x_2 \geq 1] = \mathbb{I}[-x_1 - x_2 \leq -1]$$

- **D-dimensional regression:**
  - single layer can approximate arbitrary linear function
    - 2-layer network can model indicator function of arbitrary convex polyhedron
    - 3-layer network can uniformly approximate arbitrary continuous function (as weighted sum of indicators of convex polyhedra)
- So 3 layered NN can approximate all regular dependencies!

[1] How to make XOR (exclusive OR) function?

Classification

- Consider indicator activations.
- **# layers selection for classification:**
  - single layer network selects arbitrary half-spaces
  - 2-layer network selects arbitrary convex polyhedron (by intersection of 1-layer outputs)
    - therefore it can approximate arbitrary convex sets
  - 3-layer network selects (by union of 2-layer outputs) arbitrary finite sets of polyhedra
- So 3 layered NN can approximate all regular sets!

Number of layers selection

- Why use more than 3 layers?
- 3 layered net can approximate any regular dependency, but may require too many neurons.
- Deeper neurons model more complicated dependencies.
- Using such neurons we can model the function with fewer total neurons.
    - less parameters=>less overfitting.

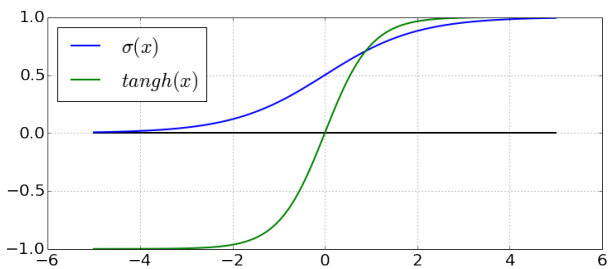
Table of Contents

Continuous activations

- $\mathbb{I}[w^T x - w_0 \leq 0]$ - piecewise constant, derivative=0, cannot optimize weights.
- Replace $\mathbb{I}[w^T x - w_0 \leq 0]$ with smooth activation $\phi(w^T x - w_0)$.

Typical activation functions

- sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
  - 1-layer neural network with sigmoidal activation is equivalent to logistic regression
- hyperbolic tangent: $\text{tangh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$
  - if $\mathbb{E}x = 0$, then $\mathbb{E}\,\text{tangh}(x) = 0$.



- Problem $\phi'(x) \approx 0$ outside (-3,3).

Activation functions

- Rectified linear unit (ReLU): $\phi(x) = \max(0, x)$
  - analog with smooth derivative (SoftPlus): $\phi(x) = \ln(1 + e^x)$
- Leaky ReLU: $\phi(x) = \begin{cases} x, & x \geq 0 \\ 0.01x, & x < 0 \end{cases}$
- Parametric ReLU (α is estimated): $\phi(x|\alpha) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$
- Exponential LU (α is fixed): $\phi(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$

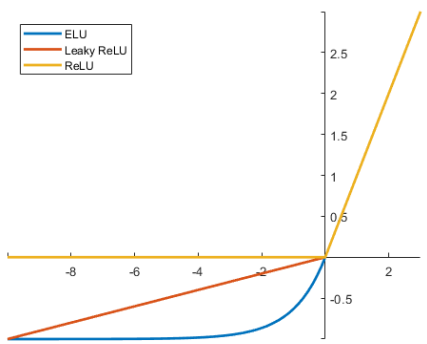


Table of Contents

Output layer and loss: regression

- Regression: $\phi(I) = I$
- Scalar regression $y \in \mathbb{R}$:

$$MSE(x, y) = (\widehat{y}(\mathbf{x}) - y)^2$$

- Vector regression $\mathbf{y} \in \mathbb{R}^K$:

$$MSE(x, y) = \|\widehat{\mathbf{y}}(\mathbf{x}) - \mathbf{y}\|_2^2$$

Output layer and loss: classification, probabilities

- **Binary classification**: $y \in \{0, 1\}$

$$p(y = +1|x) = \frac{1}{1 + e^{-I}}$$

$$\mathcal{L}(x, y) = -\ln p(y|x) = -\ln p(y = 1|x)^y [1 - p(y = 1|x)]^{1-y}$$

- **Multiclass classification**: $y \in 1, 2, ...C$

$$\{SoftMax(I_1, ...I_C)\}_j = p(y = j|x) = \frac{e^{I_j}}{\sum_{k=1}^{C} e^{I_k}}, j = 1, 2, ...C$$

$$\mathcal{L}(x, y) = -\ln p(y|x) = -\ln \prod_{c=1}^{C} p(y = c|x)^{y_c}, \qquad y_c = \mathbb{I}[y = c]$$

Output layer and loss: classification, scores

- **Binary classification**: $y \in \{0, 1\}$, $\mu > 0$ - parameter

$$O(x) = \text{preference to positive class vs. zero class}$$

$$hinge(x, y) = [\mu - yO(x)]_+$$

- **Multiclass classification**: $y \in 1, 2, ...C$, $\mu > 0$ - parameter:

$$\{O_1(x), ...O_C(x)\} \text{ - scores of classes } 1, ...C$$

$$hinge_1(x, y) = \left[\max_{c \neq y} O_c + \mu - O_y\right]_+$$

$$hinge_2(x, y) = \sum_{c \neq y} [O_c + \mu - O_y]_+$$

Special applications

Neural nets may be trained to predict not points, but probability densities:

- predict K values for K bins of the histogram (+SoftMax)
- predict parameters of a probability distribution, e.g. μ, Σ for $y|x \sim \mathcal{N}(\mu(x), \Sigma(x))$.

Autoencoder allows reproducing the input. Possible applications:

- extract informative features, dimensionality reduction
- construct initialization for first layers of deep network.
- outlier extraction[2]

[2] how?

Table of Contents

Neural network optimization

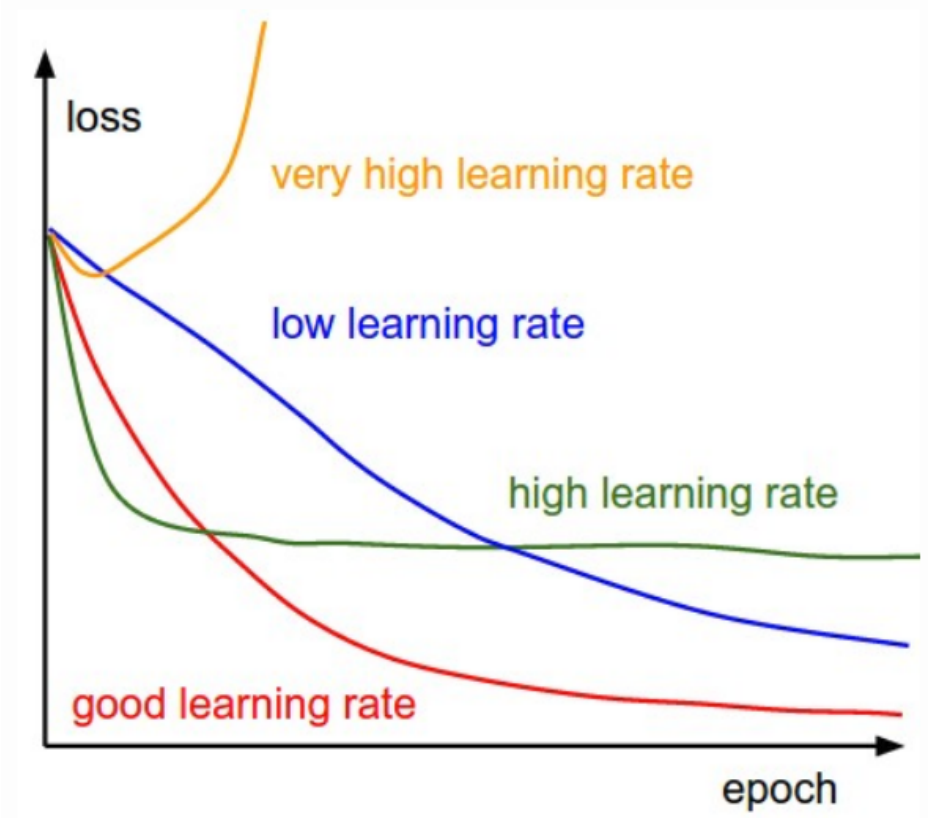
- Denote $\mathcal{L}(\widehat{y}, y)$ - the loss function, $W = \#$[weights in the network], ε - step size.
- We may optimize neural network using SGD:

```
initialize randomly w # small values for sigmoid and tangh

while not (stop condition):
    sample random object (x_i, y_i)
    w := w − ε∇_w L(w, x_i, y_i)
```

- Standardization of features makes GD & SGD converge faster

Learning rate selection is important



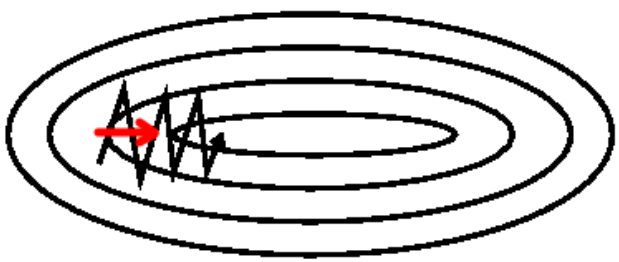
- Learning rate should gradually decrease for SGD.
- Often take const ε and decrease it after reaching plato region.

SGD with momentum

```
initialize randomly w # small values for sigmoid and tangh

while not (stop condition):
    sample random object (x_i, y_i)
    Δw := αΔw + (1 − α) ∇_w L(w, x_i, y_i) # alpha is typically 0.9.
    w := w − εΔw
```

- Converges faster, because
  - gradient is based on several observations instead of one
  - gradient directions aggregation averages ininformative trembling:



Gradient calculation

- Direct $\nabla\mathcal{L}(w)$ calculation, using

$$\frac{\partial\mathcal{L}}{\partial w_i} = \frac{\mathcal{L}(w+\varepsilon_i) - \mathcal{L}(w)}{\varepsilon} + O(\varepsilon) \tag{1}$$

or better

$$\frac{\partial\mathcal{L}}{\partial w_i} = \frac{\mathcal{L}(w+\varepsilon_i) - \mathcal{L}(w-\varepsilon_i)}{2\varepsilon} + O(\varepsilon^2) \tag{2}$$

has complexity $O(W^2)$

  - need to calculate W derivatives
  - complexity for each derivative: 2W

Backpropagation algorithm needs only $O(W)$ to evaluate all derivatives.

Gradient calculation example

- Consider binary classification $y \in \{+1, -1\}$, network predicts $p(y = +1|x)$:

$$p(y = +1|x) = \frac{1}{1 + e^{-w^T x}}$$

- Quality is evaluated with score

$$S = p(y|x) = \mathbb{I}[y = +1]p(y = +1|x) + \mathbb{I}[y = -1](1 - p(y = +1|x)$$

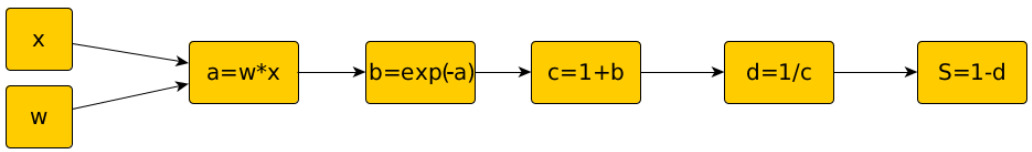
- Suppose $y = -1$ and we want to adjust w, so that $1 - p(y = +1|x)$ is higher:

$$w := w + \varepsilon \frac{\partial S}{\partial w} = w + \varepsilon \frac{\partial [1 - p(y = +1|x)]}{\partial w}$$

Backpropagation

$$\frac{\partial[1-p(y=+1|x)]}{\partial w}=\frac{\partial}{\partial w}\left[1-\frac{1}{1+e^{-w^Tx}}\right]-?$$

- Represent computations with computational graph:



- Calculate all intermediate variables left to right, then derivatives right to left:

$$\frac{\partial S}{\partial d}=-1,\ \frac{\partial S}{\partial c}=\frac{\partial S(d(c))}{\partial c}=\frac{\partial S}{\partial d}\frac{\partial d}{\partial c}=\frac{\partial S}{\partial d}\left(-\frac{1}{c^2}\right);\ \frac{\partial S}{\partial b}=$$
$$\frac{\partial S(c(b))}{\partial b}=\frac{\partial S}{\partial c}\frac{\partial c}{\partial b}=\frac{\partial S}{\partial b}\cdot 1;\ \frac{\partial S}{\partial a}=\frac{\partial S(b(a))}{\partial a}=\frac{\partial S}{\partial b}\frac{\partial b}{\partial a}=-\frac{\partial S}{\partial b}e^{-a}$$
$$\frac{\partial S}{\partial w}=\frac{\partial S(a(w))}{\partial w}=\frac{\partial S}{\partial a}\frac{\partial a}{\partial w}=\frac{\partial S}{\partial a}x$$

Features of neural network optimization

- Dependency $\widehat{y}(x)$ is non-convex in general.
- $\mathcal{L}(\widehat{y}, y)$ - non-convex => many local optima.
- Found local optimum is affected by:
  - initialization (see Xavier initialization and other)
  - objects sampled at minibatches
  - optimization method and ε_t dynamics
- Can fit the model using different methods and then
  - select best solution on the validation set
  - average several solutions

Table of Contents

Deep learning motivation

- Machine learning algorithms rely on good features
- Deep learning=neural nets with many layers.
- Neurons at deep layers can automatically learn complex features.
  - important in such domains as image, sound.
  - images: extract borders, than primitive figures, than more complex figures (eyes, tails, ...)
  - works better than predefined features

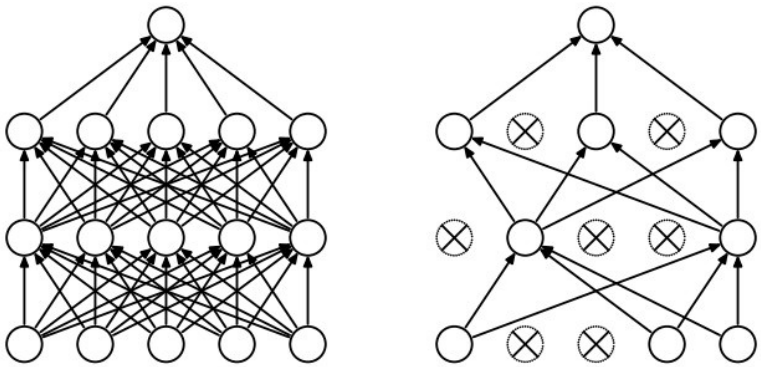
Deep learning complexity

- Deep learning models have much more parameters. Solutions to this:
  - pretrain initial layers from large available datasets with similar task
    - e.g. ImageNet (>1 mln. of labeled images) for image classification
  - use data augmentation:
    $(x, y) \rightarrow \{(\phi_1(x), y), (\phi_2(x), y), (\phi_3(x), y), ...\}$
    - $\phi_1, \phi_2, \phi_3, ...$ - some target invariant transformations
    - e.g. changes of brightness/contrast of images.

Complexity of MLP

- In MLP consider adjacent layers with N and M neurons.
  - $N \times M$ connections, so $N \times M$ weights to estimate!
- Solutions to overparametrization:
  - use fewer neurons
  - apply constraints on weights (e.g. weight sharing)
    - e.g. in convolutions
  - leave only most important connections, drop other
    - e.g. in convolutions
  - use regularization to reduce model flexibility
    - L_1, L_2
    - early stopping
    - DropOut
    - BatchNorm

DropOut: training



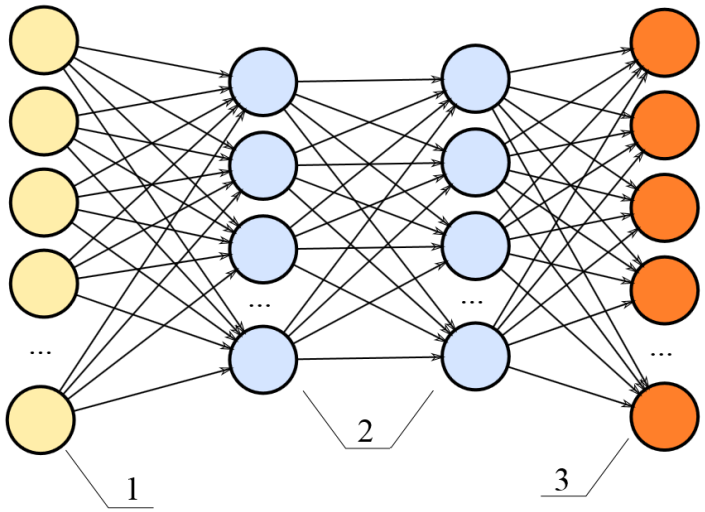
(a) Standard Neural Net (b) After applying dropout.

- For each minibatch every neuron, except output neurons, is dropped with probability $(1-p)$ and retained with probability p.
  - independently from other neurons
- This reduces ovefitting:
  - prevents co-adaptation of neurons (overfitting)
  - we train an ensemble of thinned (more simple) neural nets.
- Next neuron receives $\sum_i (pw_i x_i + (1-p)w_i 0) = \sum_i pw_i x_i$.

DropOut: inference

- every neuron produces something reasonable, so every neuron is left
- to compensate for change in architecture output of every neuron is multiplied by p
  - need to perform extra multiplications at inference, to omit them:
    - training: output x_i/p when x_i is retained
    - inference: output every neuron x_i as is.

BatchNorm: motivation



- SGD $w := w - \varepsilon \nabla_w \mathcal{L}(x, y)$ updates all weights on all layers simultaneously.
- Distribution of outputs changes and later layers have to relearn from scratch.
- Also input can shift to saturation region of non-linearity.

BatchNorm: intro

- Idea: standardize outputs at intermediate layers

$$\tilde{x}_k = \frac{x_k - \mu_k}{\sigma_k}, \quad \mu_k = \mathbb{E}x_k, \sigma_k = \sqrt{Var(x_k)}$$

  - guarantees $\mathbb{E}\tilde{x}_k = 0,\ \text{Var}\,\tilde{x}_k = 1$ after weight updates on previous layers.
    - training is faster for later layers
- Training:
  - problem: don't know μ_k, σ_k
    - they change dynamically with weight updates
  - solution: estimate them on current minibatch (should be large enough)
- Inference:
  - since now distribution of x_k is fixed, can estimate μ_k, σ_k from the whole training set.
  - more efficient: average estimates of μ_k, σ_k from final minibatches of training.

BatchNorm: actual version

$$\tilde{x}_k = \alpha_k \frac{x_k - \mu_k}{\sqrt{\sigma_k^2 + \varepsilon}} + \beta_k, \quad \mu_k = \bar{x}_k,\ \sigma_k = \sqrt{Var(x_k)},\ \varepsilon = 10^{-6}.$$

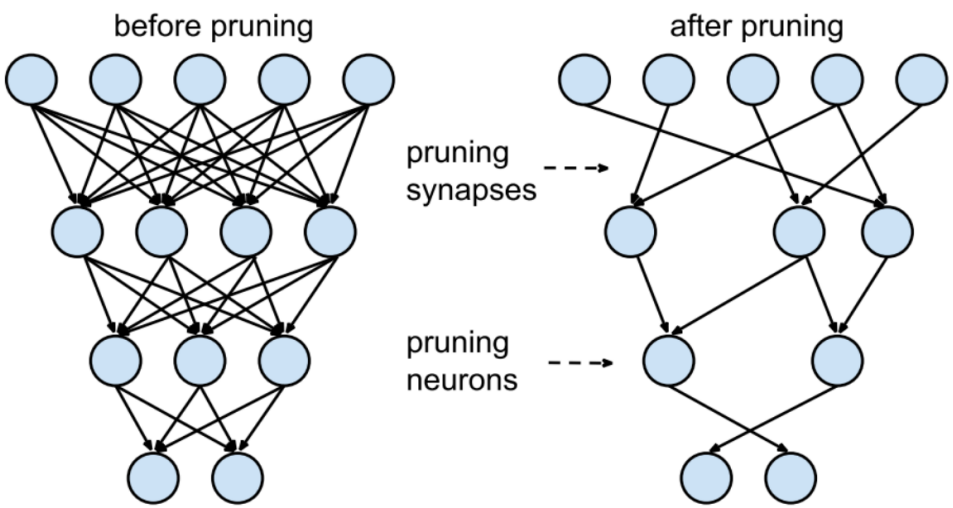
- Training:
  - μ_k, σ_k estimated from the minibatch
  - α_k, β_k - output std.dev. and mean.
    - learned during backpropagation
  - motivation:
    - may cancel normalization effect (e.g. in image->time prediction)
    - flexibility to adjust better to non-linearity
- Inference:
  - μ_k, σ_k estimated fixed to be mean, std.dev. from a wide set of objects.
  - α_k, β_k fixed.

Reducing computational complexity

To reduce computational complexity train a large network, and then thin it out:

- drop connections with low weight
- discard neurons with low input weights
- discard neurons with weak activations

Pruning of neurons and connections



Optimal brain damage

$$L(w) - L(w^*) =$$

$$= \nabla L(w^*)^T (w - w^*) + \frac{1}{2} (w - w^*)^T \nabla^2 L(w^*) (w - w^*) + O\left(\|w - w^*\|^3\right)$$

$$\approx \frac{1}{2} (w - w^*)^T \nabla^2 L(w^*) (w - w^*) \approx \frac{1}{2} \sum_i \frac{\partial L(w^*)}{\partial w_i^2} (w_i - w_i^*)^2$$

Approximations:

- $O\left(\|w - w^*\|^3\right) \approx 0$
- we converged to exact optimum, so $\nabla L(w^*) = 0$
- $\nabla^2 L(w^*)$ is diagonal (ignore mixed derivatives effect)

Optimal brain damage algorithm:

- repeat:
  - train network
  - drop by $\frac{\partial L(w^*)}{\partial w_i^2} (w_i - w_i^*)^2 < threshold$

Conclusion

Properties of neural networks:

- Universal approximator: can model complex non-linear relationships.
- Deep nets can automatically construct intelligent features.
  - work better than hand-made
- Loss function is highly non-linear and non-convex.
- Everything in optimization affects final result:
  - starting conditions, optimization algorithm, learning rate size, its decrease schedule, minibatch size.
- Have much more parameters than traditional ML models.
  - drop neurons/connections
  - impose constraints on weights (e.g. weight sharing)
  - use regularization: early stopping, L1/L2, DropOut, BatchNorm.