Outline

1 Neural Network architecture
   ● Neurons
   ● A historical model/algorithm - the perceptron
   ● Going beyond perceptron - multilayer neural networks
   ● Neural network training

2 Hyperparameters
   ● How to choose the number of hidden layers/neurons?
   ● Activation functions
   ● Output units
   ● Loss functions
   ● Weight initialization

3 Regularization
   ● Penalization
   ● Dropout
   ● Batch normalization
   ● Early stopping

4 All in all
What to set in a neural network?

Network structure:
- Number of layers/neurons per layer
- Activation functions
- Output unit
- Specific layers (dropout, batch normalization)

Optimization:
- Optimization algorithm
- Weights/biases initialization
- Loss function
What to set in a neural network?

Network structure:
- Number of layers/neurons per layer
- Activation functions
- Output unit
- Specific layers (dropout, batch normalization)

Optimization:
- Loss function
- Optimization algorithm
- Weights/biases initialization
Outline

1 Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

2 Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization

3 Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

4 All in all
Number of hidden layers/neurons

- No particular rules for choosing the number of layers or the number of neurons per layer.

- Read research papers related to the task you want to solve and test the architecture they propose.

- You may want to change the architecture a bit to see how it influences the performance.

- Beware: there exist many rules of thumbs which are not supported by evidence (either practical or theoretical).
Number of hidden layers/neurons

- Use **data-driven strategies:**
  - Network **pruning** following the procedure **training/pruning/training/pruning/...**
    - [“What is the state of neural network pruning?”, Blalock et al. 2020]
  - More complex **evolutionary algorithms**
    - [“AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data”, Egele et al. 2020]
Outline

1 Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

2 Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
     - Output units
     - Loss functions
     - Weight initialization

3 Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

4 All in all
Sigmoid activation function

![Sigmoid activation function graph]

Figure: Sigmoid activation function $\sigma$

$$\sigma : x \mapsto \frac{\exp(x)}{1 + \exp(x)}$$

- Saturated function due to horizontal asymptotes:
  - Gradient is close to zero in these two areas ($\pm \infty$)
  - Rescaling the inputs of each layer can help to avoid these areas.
Sigmoid activation function

\[ \sigma : x \mapsto \frac{\exp(x)}{1 + \exp(x)} \]

- Sigmoid is not a zero-centered function
  - Rescaling data
- Computing \( \exp(x) \) is a bit costly
Hyperbolic tangent

Figure: Hyperbolic tangent (tanh)

\[
\text{tanh} : x \mapsto \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}
\]

- The function tanh is zero-centered
  - No need for rescaling data
Hyperbolic tangent

\[
\tanh : x \mapsto \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}
\]

- **Saturated function due to horizontal asymptotes:**
  - Gradient is close to zero in these two areas \((\pm \infty)\)
  - Rescaling the inputs of each layer can help to avoid these areas.
Hyperbolic tangent

Figure: Hyperbolic tangent (tanh)

\[
tanh : x \mapsto \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}
\]

- Computing \(\exp(x)\) is a bit costly
- Note that \(\tanh(x) = 2\sigma(2x) - 1\)
Rectified Linear Unit (ReLU)

\[ \text{ReLU} : x \mapsto \max(0, x) \]

- Not a saturated function in \(+\infty\)
- But saturated/null in the region \(x \leq 0\)
- Computationally efficient
- Training NN with ReLU is faster than with sigmoid/tanh.
- Biologically plausible
More on ReLU


Figure: Good parameter initialization - ReLU is active
More on ReLU


Figure: Bad parameter initialization - ReLU outputs zero
More on ReLU

Figure: Bad parameter initialization - ReLU outputs zero

ReLU output can be zero but positive initial bias can help.


- Most of the time, neurons are inactive.
- when they activate, their activation is proportional to their input.
Parametric ReLU

\[ x \mapsto \max(\alpha x, x) \]

- **Leaky ReLU**: \( \alpha = 0.1 \)
  
  ["Rectifier nonlinearities improve neural network acoustic models", Maas et al. 2013]
Parametric ReLU

\text{Figure: Parametric ReLU}

Parametric ReLU: \( x \mapsto \max(\alpha x, x) \)

- \textbf{Absolute Value Rectification:}
  \[
  \alpha = -1
  \]

[“What is the best multi-stage architecture for object recognition?”, Jarrett et al. 2009]
Parametric ReLU

Figure: Parametric ReLU

Parametric ReLU: $x \mapsto \max(\alpha x, x)$

- **Parametric ReLU**: $\alpha$ optimized during backpropagation. Activation function is learned.

["Empirical evaluation of rectified activations in convolutional network", Xu et al. 2015]
Exponential Linear Unit (ELU)

\[
ELU: x \mapsto \begin{cases} 
  x & \text{if } x \geq 0 \\
  \alpha (\exp(x) - 1) & \text{otherwise}
\end{cases}
\]

- Close to ReLU but differentiable
- Closer to zero mean output.
- \(\alpha\) is set to 1.0.
- Robustness to noise

["Fast and accurate deep network learning by exponential linear units (elus)", Clevert et al. 2015]
Maxout

Figure: Maxout activation function, with $k = 3$ pieces

$$x \mapsto \max(w_1 x + b_1, w_2 x + b_2, w_3 x + b_3)$$

- Learn piecewise linear functions with $k$ pieces: no saturation.
**Maxout**

Figure: Maxout activation function, with $k = 3$ pieces

$$x \mapsto \max(w_1 x + b_1, w_2 x + b_2, w_3 x + b_3)$$

- **Number of parameters multiplied by $k$**

- **Resist to catastrophic forgetting**
Swish

Figure: Swish function for $\beta = 0.1, 1, 10$

$$Swish : x \mapsto x \frac{\exp(\beta x)}{1 + \exp(\beta x)}$$

- Swish interpolates between the linear function and ReLU.
- [“Searching for activation functions”, Ramachandran et al. 2017]
- Non-monotonic function - seems to be an important feature.
Conclusion on activation functions

- Use ReLU (or Swish).
- Test Leaky ReLU, maxout, ELU.
- Try out Tanh, but do not expect too much.
- Do not use sigmoid.
Outline

1 Neural Network architecture
   • Neurons
   • A historical model/algorithm - the perceptron
   • Going beyond perceptron - multilayer neural networks
   • Neural network training

2 Hyperparameters
   • How to choose the number of hidden layers/neurons?
   • Activation functions
   • Output units
   • Loss functions
   • Weight initialization

3 Regularization
   • Penalization
   • Dropout
   • Batch normalization
   • Early stopping

4 All in all
Output units

- **Linear output unit:**

  \[ \hat{y} = W^T h + b \]

  → Linear regression based on the new variables \( h \).
Output units

- **Sigmoid output unit**, used to predict \{0, 1\} outputs:

\[
P(Y = 1|h) = \sigma(W^T h + b),
\]
where \(\sigma(t) = e^t/(1 + e^t)\).

→ Logistic regression based on the new variables \(h\).

- **Softmax output unit**, used to predict \{1, ..., K\}:

\[
\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}
\]
where, each \(z_i\) is the activation of one neuron of the previous layer, given by \(z_i = W_i^T h + b_i\).

→ Multinomial logistic regression based on the new variables \(h\).
Multinomial logistic regression

Generalization of logistic regression for multiclass outputs: for all $1 \leq k \leq K$,

$$\log \left( \frac{\mathbb{P}[Y_i = k]}{Z} \right) = \beta_k X_i, \quad (5)$$

Hence, for all $1 \leq k \leq K$,

$$\mathbb{P}[Y_i = k] = \frac{Z e^{\beta_k X_i}}{\sum_{k=1}^{K} e^{\beta_k X_i}}, \quad (6)$$

where

$$Z = \frac{1}{\sum_{k=1}^{K} e^{\beta_k X_i}}. \quad (7)$$

Thus,

$$\mathbb{P}[Y_i = k] = \frac{e^{\beta_k X_i}}{\sum_{\ell=1}^{K} e^{\beta_{\ell} X_i}}. \quad (8)$$
Biology bonus

Softmax, used with cross-entropy:

\[- \log P(Y = y | z) \quad (9)\]

\[= - \log \text{softmax}(z)_y \quad (10)\]

\[= - z_y + \log \left( \sum_j e^{z_j} \right) \quad (11)\]

\[\simeq \max_j z_j - z_y , \quad (12)\]
Biology bonus

Softmax, used with cross-entropy:

\[- \log(\mathbb{P}(Y = y|z)) \quad (9)\]
\[= - \log \text{softmax}(z)_y \quad (10)\]
\[= - z_y + \log \left( \sum_j e^{z_j} \right) \quad (11)\]
\[\simeq \max_j z_j - z_y, \quad (12)\]

No contribution to the cost when \(\text{softmax}(z)_{\hat{y}}\) is maximal.
Softmax, used with cross-entropy:

\[- \log (P(Y = y | z)) = - \log \text{softmax}(z)_y\]

\[= - z_y + \log \left( \sum_j e^{z_j} \right) \]

\[\simeq \max_j z_j - z_y,\]

No contribution to the cost when \(\text{softmax}(z)_{\hat{y}}\) is maximal.

**Lateral inhibition:** believed to exist between nearby neurons in the cortex. When the difference between the max and the other is large, winner takes all: one neuron is set to 1 and the others go to zero.
Biology bonus

Softmax, used with cross-entropy:

\[- \log(\mathbb{P}(Y = y|z)) \quad (9)\]
\[= - \log \text{softmax}(z)_y \quad (10)\]
\[= - z_y + \log \left( \sum_{j} e^{z_j} \right) \quad (11)\]
\[\simeq \max_{j} z_j - z_y, \quad (12)\]

No contribution to the cost when \(\text{softmax}(z)_{\hat{y}}\) is maximal.

More complex models: Conditional Gaussian Mixture: \(Y\) is multimodal [“On supervised learning from sequential data with applications for speech recognition”; “Generating sequences with recurrent neural networks”, Schuster 1999; Graves 2013].
Outline

1. Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization

3. Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

4. All in all
Cost functions

- **Mean Square Error (MSE)**

\[
\frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f_\theta(X_i)) = \frac{1}{n} \sum_{i=1}^{n} (Y_i - f_\theta(X_i))^2
\]

- **Mean Absolute Error**

\[
\frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f_\theta(X_i)) = \frac{1}{n} \sum_{i=1}^{n} |Y_i - f_\theta(X_i)|
\]

- **0 − 1 Error**

\[
\frac{1}{n} \sum_{i=1}^{n} \ell(Y_i, f_\theta(X_i)) = \frac{1}{n} \sum_{i=1}^{n} 1_{Y_i \neq f_\theta(X_i)}
\]
Cost functions

Cross entropy (or negative log-likelihood):

\[ \ell(y_i, f_\theta(x_i)) = -\log \left( [f_\theta(x_i)]_{y_i} \right) \quad (13) \]

- Very popular!
Cost functions

Cross entropy (or negative log-likelihood):

\[ \ell(y_i, f_\theta(x_i)) = -\log \left( [f_\theta(x_i)]_{y_i} \right) \quad (13) \]

- Should help to prevent saturation:

\[ -\log(P(Y = y_i | X = x_i)) \quad (14) \]

\[ = -\log(\sigma((2y - 1)(W^T h + b))), \]

with

\[ \sigma(t) = \frac{e^t}{1 + e^t} \]

Usually, saturation occurs when \((2y - 1)(W^T h + b) \ll -1\). In this case, \(-\log(P(Y = y_i | X))\) is linear in \(W\) and \(b\), therefore preventing saturation to happen.
Cost functions

Cross entropy (or negative log-likelihood):

\[ \ell(y_i, f_\theta(x_i)) = -\log (f_\theta(x_i)_{y_i}) \] (13)

- Mean Square Error should not be used with softmax output units

[“Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition”, Bridle 1990]
Outline

1 Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

2 Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization

3 Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

4 All in all
Weight initialization

**First idea:** Set all weights and bias to the same value.

*WeKnowMemes*

When you initialise your ML noob friend’s NN weights with zeros

*That’s the evilest thing I can imagine.*
Small or big weights?

Consider the initial weight distribution to be $\mathcal{N}(0, \sigma^2)$.

1. If the variance of the weights is too small, that is $\sigma^2 \ll 1$:
   - the output of each neuron is close to a Dirac in 0: there is no activation at all.

2. If the variance of the weights is too large, that is $\sigma^2 \gg 1$:
   - the linear combinations are very large, which increases the saturation phenomenon.

3. In any case, no need to tune the bias: they can be initially set to zero.
Other initialization

**Idea:** the variance of the input should be the same as the variance of the output.

Let $w_j$ be any weight between layer $j$ and layer $j + 1$.

**He et al. initialization**

[“Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”, He et al. 2015]

Initialize bias to zero and weights randomly using

$$w_j \sim \mathcal{N} \left(0, \frac{\sqrt{2}}{n_j} \right),$$

where $n_j$ is the size of layer $j$. 

Other initialization

**Idea:** the variance of the input should be the same as the variance of the output.

Let $w_j$ be any weight between layer $j$ and layer $j+1$.

**Xavier initialization**

[“Understanding the difficulty of training deep feedforward neural networks”, Glorot and Bengio 2010]

Initialize bias to zero and weights randomly using

$$w_j \sim \mathcal{U} \left[ -\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}} \right],$$

where $n_j$ is the size of layer $j$

→ Not theoretically valid for ReLU
Other initialization

**Idea:** the variance of the input should be the same as the variance of the output.

Let $w_j$ be any weight between layer $j$ and layer $j+1$.

### Xavier initialization

[“Understanding the difficulty of training deep feedforward neural networks”, Glorot and Bengio 2010]

Initialize bias to zero and weights randomly using

$$w_j \sim \mathcal{U} \left[ -\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}} \right],$$

where $n_j$ is the size of layer $j$

→ Not theoretically valid for ReLU

**Bonus:** [“All you need is a good init”, Mishkin and Matas 2015]
Outline

1 Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

2 Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization

3 Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

4 All in all
Regularizing to avoid overfitting

Avoid overfitting by imposing some constraints over the parameter space.

Reducing variance and increasing bias.
Avoiding overfitting

- **Penalization (L1 or L2)**
  Replacing the cost function $\mathcal{L}$ by
  \[
  \tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta, X, y) + \text{pen}(\theta).
  \]

- **Soft weight sharing - see CNN lecture**
  Reduce the parameter space artificially by imposing explicit constraints.

- **Dropout**
  Randomly kill some neurons during optimization and predict with the full network.

- **Batch normalization**
  Renormalize a layer inside a batch, so that the network does not overfit on this particular batch.

- **Early stopping**
  Stop the gradient descent procedure when the error on the validation set increases.
Outline

1 Neural Network architecture
   ● Neurons
   ● A historical model/algorithm - the perceptron
   ● Going beyond perceptron - multilayer neural networks
   ● Neural network training

2 Hyperparameters
   ● How to choose the number of hidden layers/neurons?
   ● Activation functions
   ● Output units
   ● Loss functions
   ● Weight initialization

3 Regularization
   ● Penalization
   ● Dropout
   ● Batch normalization
   ● Early stopping

4 All in all
Constraint the optimization problem

\[
\min_{\theta} \mathcal{L}(\theta, X, y), \quad \text{s.t.} \quad \text{pen}(\theta) \leq \text{cste}. \quad (14)
\]

Using Lagrangian formulation, this problem is equivalent to:

\[
\min_{\theta} \mathcal{L}(\theta, X, y) + \lambda \text{pen}(\theta), \quad (15)
\]

where

- \( \mathcal{L}(\theta, X, y) \) is the loss function (data-driven term)
- \( \text{pen} \) is a function that increases when \( \theta \) becomes more complex (penalty term)
- \( \lambda \geq 0 \) is a constant standing for the strength of the penalty term.

For Neural Networks, \( \text{pen} \) only penalizes the weights and not the bias: the latter being easier to estimate than weights.
Example of penalization

\[ \min_{\theta} \mathcal{L}(\theta, X, y) + \text{pen}(\theta), \]

- **Ridge**

\[ \text{pen}(\theta) = \lambda \|\theta\|^2 \]


[“Lecture notes on ridge regression”, Wieringen 2015]
Example of penalization

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \text{pen}(\theta),$$

- **Lasso**

  $$\text{pen}(\theta) = \lambda \|\theta\|_1$$

["Regression shrinkage and selection via the lasso",
Tibshirani 1996]
Example of penalization

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \text{pen}(\theta),$$

- **Elastic Net**

  $$\text{pen}(\theta) = \lambda \|\theta\|_2^2 + \mu \|\theta\|_1$$

[“Regularization and variable selection via the elastic net”, Zou and Hastie 2005]
Simple case: linear regression

Linear regression

The estimate of linear regression \( \hat{\beta} \) is given by

\[
\hat{\beta} \in \arg\min_{\beta \in \mathbb{R}^d} \sum_{i=1}^{n} \left( Y_i - \sum_{j=1}^{d} \beta_j x_i^{(j)} \right)^2,
\]

which can be written as

\[
\hat{\beta} \in \arg\min_{\beta \in \mathbb{R}^d} \| Y - X \beta \|_2^2,
\]

where \( X \in M_{n,d}(\mathbb{R}) \).

Solution:

\[
\hat{\beta} = (X'X)^{-1}X'Y.
\]
Penalized regression

Penalized linear regression

The estimate of linear regression $\hat{\beta}_{\lambda,q}$ is given by

$$\hat{\beta}_{\lambda,q} \in \arg\min_{\beta \in \mathbb{R}^d} \| Y - X\beta \|_2^2 + \lambda \| \beta \|_q^q.$$

- $q = 2$: Ridge linear regression
- $q = 1$: LASSO
Ridge regression, $q = 2$

Ridge linear regression

The ridge estimate $\hat{\beta}_{\lambda, 2}$ is given by

$$\hat{\beta}_{\lambda, 2} \in \arg\min_{\beta \in \mathbb{R}^d} \| Y - X\beta \|_2^2 + \lambda \| \beta \|_2^2.$$

Solution:

$$\hat{\beta}_{\lambda, 2} = (X'X + \lambda I)^{-1}X'Y. \quad (19)$$

This estimate has a bias equal to $-\lambda (X'X + \lambda I)^{-1}\beta$, and a variance $\sigma^2 (X'X + \lambda I)^{-1}X'X(X'X + \lambda I)^{-1}$. Note that

$$\mathbb{V}[\hat{\beta}_{\lambda, 2}] \leq \mathbb{V}[\hat{\beta}].$$

In the case of orthonormal design ($X'X = I$), we have

$$\hat{\beta}_{\lambda, 2} = \frac{\hat{\beta}}{1 + \lambda} = \frac{1}{1 + \lambda} X'Y. \quad (20)$$
Sparsity

There is another desirable property on $\hat{\beta}$

If $\hat{\beta}_j = 0$, then feature $j$ has no impact on the prediction:

$$\hat{y} = \text{sign}(x^T \hat{\beta} + \hat{b})$$

If we have many features ($d$ is large), it would be nice if $\hat{\beta}$ contained zeros, and many of them

- Means that only few features are statistically relevant.
- Means that only few features are useful to predict the label

Leads to a simpler model, with a “reduced” dimension

How do we enforce sparsity in $\beta$?
Sparsity

Tempting to solve

\[ \hat{\beta}_{\lambda,0} \in \arg\min_{\beta \in \mathbb{R}^d} \| Y - X\beta \|_2^2 + \lambda \| \beta \|_0. \]  

(21)

where

\[ \| \beta \|_0 = \# \{ j \in \{1, \ldots, d\} : \beta_j \neq 0 \}. \]

To solve this, explore all possible supports of \( \beta \). Too long! (NP-hard)

Find a convex proxy of \( \| \cdot \|_0 \): the \( \ell_1 \)-norm

\[ \| \beta \|_1 = \sum_{j=1}^{d} |\beta_j| \]
LASSO

Least Absolute Selection and Shrinkage Operator

Lasso linear regression

The LASSO estimate of linear regression \( \hat{\beta}_{\lambda,1} \) is given by

\[
\hat{\beta}_{\lambda,1} \in \arg\min_{\beta \in \mathbb{R}^d} \|Y - X\beta\|_2^2 + \lambda\|\beta\|_1.
\] (22)

Solution: No close form in the general case

If the \( X_j \) are orthonormal then

\[
\hat{\beta}_{\lambda,1,j} = X_j'Y \left(1 - \frac{\lambda}{2|X_j'Y|}\right)_+,
\] (23)

where \((x)_+ = \max(0, x)\).

Thus, in the very specific case of orthogonal design, we can easily show that L1 penalization implies a sparse vector if the parameter \( \lambda \) is properly tuned.
Outline

1. Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization

3. Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

4. All in all
Dropout refers to dropping out units (hidden and visible) in a neural network, i.e., temporarily removing it from the network, along with all its incoming and outgoing connections.

Each unit is independently dropped with probability

- $p = 0.5$ for hidden units
- $p \in [0, 0.5]$ for input units, usually $p = 0.2$. 

"Improving neural networks by preventing co-adaptation of feature detectors", Hinton et al. 2012
Dropout

At train time
- \( w_2 \)
- \( w_3 \)
- \( w_4 \)
- present with probability \( 1 - p \)

At test time
- \( (1 - p)w_2 \)
- \( (1 - p)w_3 \)
- \( (1 - p)w_4 \)
- always present
Dropout algorithm

**Training step.** While *not convergence*

1. Inside one epoch, for each mini-batch of size $m$,
   1. Sample $m$ different mask. A mask consists in one Bernoulli per node of the network (inner and entry nodes but not output nodes). These Bernoulli variables are *i.i.d.*
      Usually
      - the probability of selecting an hidden node is 0.5
      - the probability of selecting an input node is 0.8

2. For each one of the $m$ observation in the mini-batch,
   - Do a forward pass on the masked network
   - Compute backpropagation in the masked network
   - Compute the average gradient

3. Update the parameter according to the usual formula.

**Prediction step.**
Use all neurons in the network with weights given by the previous optimization procedure, times the probability $p$ of being selected (0.5 for inner nodes, 0.8 for input nodes).
Another way of seeing dropout - Ensemble methods

Averaging many different neural networks.
Another way of seeing dropout - Ensemble methods

Averaging many different neural networks. Different can mean either:

- randomizing the data set on which we train each network (via subsampling)
  
  **Problem:** not enough data to obtain good performance...

- building different network architectures and train each large network separately on the whole training set
  
  **Problem:** computationally prohibitive at training time and test time!

[“Fast dropout training”, Wang and Manning 2013]
[“Dropout: A simple way to prevent neural networks from over-fitting”, Srivastava et al. 2014]

**Dropping weights instead of the whole neurons:**

[“Regularization of neural networks using dropconnect”, Wan et al. 2013]
Outline

1. Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization

3. Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

4. All in all
Batch normalization

The network converges faster if its input are scaled (mean, variance) and decorrelated.

[“Efficient backprop”, LeCun et al. 1998]

Hard to decorrelate variables: requiring to compute covariance matrix.

[“Batch normalization: Accelerating deep network training by reducing internal covariate shift”, Ioffe and Szegedy 2015]

Ideas:

- Improving gradient flows
- Allowing higher learning rates
- Reducing strong dependence on initialization
- Related to regularization (maybe slightly reduces the need for Dropout)
For every neuron $k$ in the first layer, which outputs $x_i^{(k)}$ for the $i$th observation,

1. $\mu_B^{(k)} = \frac{1}{m} \sum_{i=1}^{m} x_i^{(k)}$
2. $\sigma_{B,k}^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i^{(k)} - \mu_B^{(k)})^2$
3. $\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_{B,k}^2 + \varepsilon}}$
4. $y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)} \equiv BN_{\gamma^{(k)},\beta^{(k)}}(x_i^{(k)})$

$y_i^{(k)}$ is fed to the next layer and the procedure iterates.

Backpropagation is performed on the network parameters including $(\gamma^{(k)}, \beta^{(k)})$ for all $k = 1, \ldots, H_1$, where $H_1 \in \mathbb{N}$ is the number of neurons in the first layer.

For inference, compute the average over many training batches $B$ of size $m$:

$$E_B[x^{(k)}] = E_B[\mu_B^{(k)}] \quad \text{and} \quad V_B[x^{(k)}] = \frac{m}{m-1} E_B[\sigma_{B,k}^2].$$

For inference, replace every function $x^{(k)} \mapsto BN_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ in the network by

$$x^{(k)} \mapsto \gamma \left( \frac{x^{(k)} - E_B[x^{(k)}]}{\sqrt{V_B[x^{(k)]} + \varepsilon}} \right) + \beta^{(k)}.$$

See [“Batch normalization: Accelerating deep network training by reducing internal covariate shift”, Ioffe and Szegedy 2015]
Outline

1. Neural Network architecture
   - Neurons
   - A historical model/algorithm - the perceptron
   - Going beyond perceptron - multilayer neural networks
   - Neural network training

2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization

3. Regularization
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping

4. All in all
Early stopping

Idea:
- Store the parameter values that lead to the lowest error on the validation set
- Return these values rather than the latest ones.
Early stopping algorithm

Parameters:

- patience $p$ of the algorithm: number of times to observe no improvement on the validation set error before giving up;
- the number of steps $n$ between evaluations.
Early stopping algorithm

1. Start with initial random values $\theta_0$.

2. Let $\theta^* = \theta_0$, $\text{Err}^* = \infty$, $j = 0$, $i = 0$.

3. While $j < p$
   1. Update $\theta$ by running the training algorithm for $n$ steps
   2. $i = i + n$
   3. Compute the error $\text{Err}(\theta)$ on the validation set
   4. If $\text{Err}(\theta) < \text{Err}^*$
      1. $\theta^* = \theta$
      2. $\text{Err}^* = \text{Err}(\theta)$
      3. $j = 0$
      else $j = j + 1$.

4. Return $\theta^*$ and the overall number of steps $i^* = i - np$. 
How to leverage on early stopping?

First idea: use early stopping to determine the best number of iterations $i^*$ and train on the whole data set for $i^*$ iterations.

Let $X^{(\text{train})}, y^{(\text{train})}$ be the training set.

- Split $X^{(\text{train})}, y^{(\text{train})}$ into $X^{(\text{subtrain})}, y^{(\text{subtrain})}$ and $X^{(\text{valid})}, y^{(\text{valid})}$.

- Run early stopping algorithm starting from random $\theta$ using $X^{(\text{subtrain})}, y^{(\text{subtrain})}$ for training data and $X^{(\text{valid})}, y^{(\text{valid})}$ for validation data. This returns $i^*$ the optimal number of steps.

- Set $\theta$ to random values again.

- Train on $X^{(\text{train})}, y^{(\text{train})}$ for $i^*$ steps.
How to leverage on early stopping?

Second idea: find the training error and the best parameters via early stopping. Starting from these parameters, train on the whole data set until the error matches the previous training error.

Let $X^{(train)}, y^{(train)}$ be the training set.

- Split $X^{(train)}, y^{(train)}$ into $X^{(subtrain)}, y^{(subtrain)}$ and $X^{(valid)}, y^{(valid)}$.
- Run early stopping algorithm starting from random $\theta$ using $X^{(subtrain)}, y^{(subtrain)}$ for training data and $X^{(valid)}, y^{(valid)}$ for validation data. This returns the optimal parameters $\theta^*$.
- Set $\varepsilon = \mathcal{L}(\theta^*, X^{(subtrain)}, y^{(subtrain)})$.
- While $\mathcal{L}(\theta^*, X^{(valid)}, y^{(valid)}) > \varepsilon$, train on $X^{(train)}, y^{(train)}$ for $n$ steps.
To go further

- **Early stopping is a very old idea**
  - [“Three topics in ill-posed problems”, Wahba 1987]
  - [“A formal comparison of methods proposed for the numerical solution of first kind integral equations”, Anderssen and Prenter 1981]
  - [“Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping”, Caruana et al. 2001]

- **But also an active area of research**
  - [“Adaboost is consistent”, Bartlett and Traskin 2007]
  - [“Boosting algorithms as gradient descent”, Mason et al. 2000]
  - [“On early stopping in gradient descent learning”, Yao et al. 2007]
  - [“Early stopping for kernel boosting algorithms: A general analysis with localized complexities”, Wei et al. 2017]
More on reducing overfitting

- **Soft-weight sharing:**
  
  [“Simplifying neural networks by soft weight-sharing”, Nowlan and Hinton 1992]

- **Model averaging:**
  
  Average over: random initialization, random selection of minibatches, hyperparameters, or outcomes of nondeterministic neural networks.

- **Boosting neural networks by incrementally adding neural networks to the ensemble**
  

- **Boosting has also been applied interpreting an individual neural network as an ensemble, incrementally adding hidden units to the networks**
  
  [“Convex neural networks”, Bengio et al. 2006]
Outline

1 Neural Network architecture
   • Neurons
   • A historical model/algorithm - the perceptron
   • Going beyond perceptron - multilayer neural networks
   • Neural network training

2 Hyperparameters
   • How to choose the number of hidden layers/neurons?
   • Activation functions
   • Output units
   • Loss functions
   • Weight initialization

3 Regularization
   • Penalization
   • Dropout
   • Batch normalization
   • Early stopping

4 All in all
Pipeline for neural networks

- Step 1: Preprocess/normalize the data.
- Step 2: Choose the NN architecture (number of layers, number of nodes per layer...)
- Step 3: train the network
- Step 4: Find the best learning rate (LR)
  1. The error does not change too much (LR too small) or the error explodes, NaN (LR too high).
  2. Find a rough range $[10^{-5}, 10^{-3}]$.
- Sanity checks:
  1. Compare the NN loss to that of a dummy classifier.
  2. Increasing regularization should increase the training set error
  3. A NN trained on a small fraction of the data should overfit.

Playing with neural network:

http://playground.tensorflow.org/


