# Lecture 5 - Training & Regularizing Neural Networks

DD2424

April 5, 2017

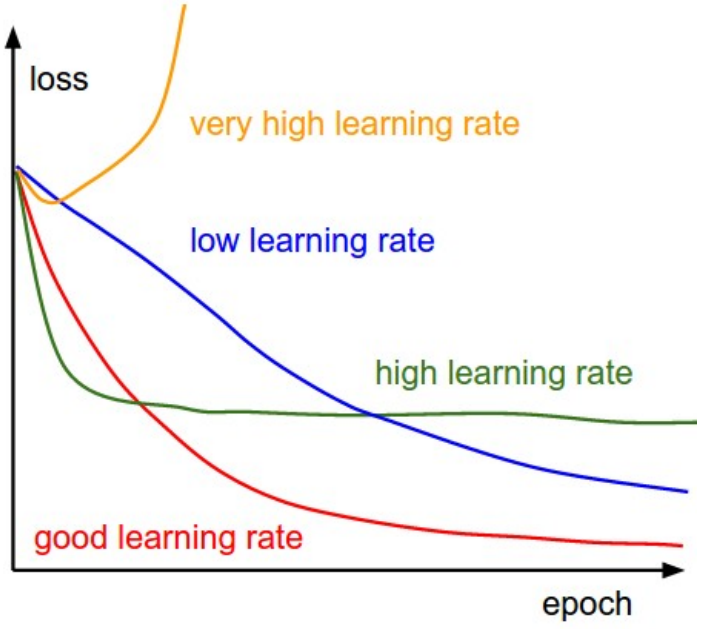
Baby sitting the training process

# Training neural networks not completely trivial

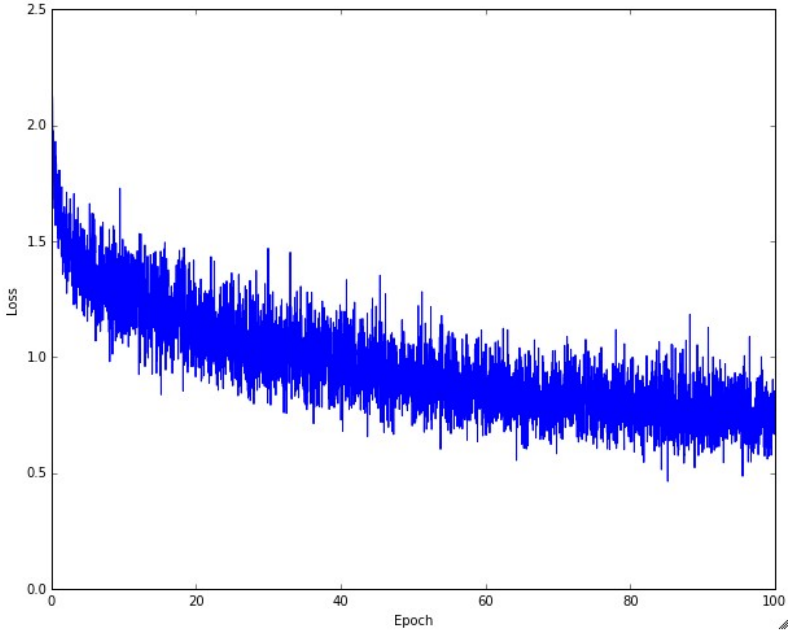
- Several **hyper-parameters** affect the quality of your training.
- These include
  - learning rate
  - degree of regularization
  - network architecture
  - hyper-parameters controlling weight initialization
- If these (potentially correlated) hyper-parameters are not appropriately set $\Longrightarrow$ you will not learn an **effective** network.
- Multiple quantities you should monitor during training.
- These quantities indicate
  - a reasonable hyper-parameter setting and/or
  - how hyper-parameters setting could be changed for the better.

# What to monitor during training

# Monitor & Visualize the loss/cost curve

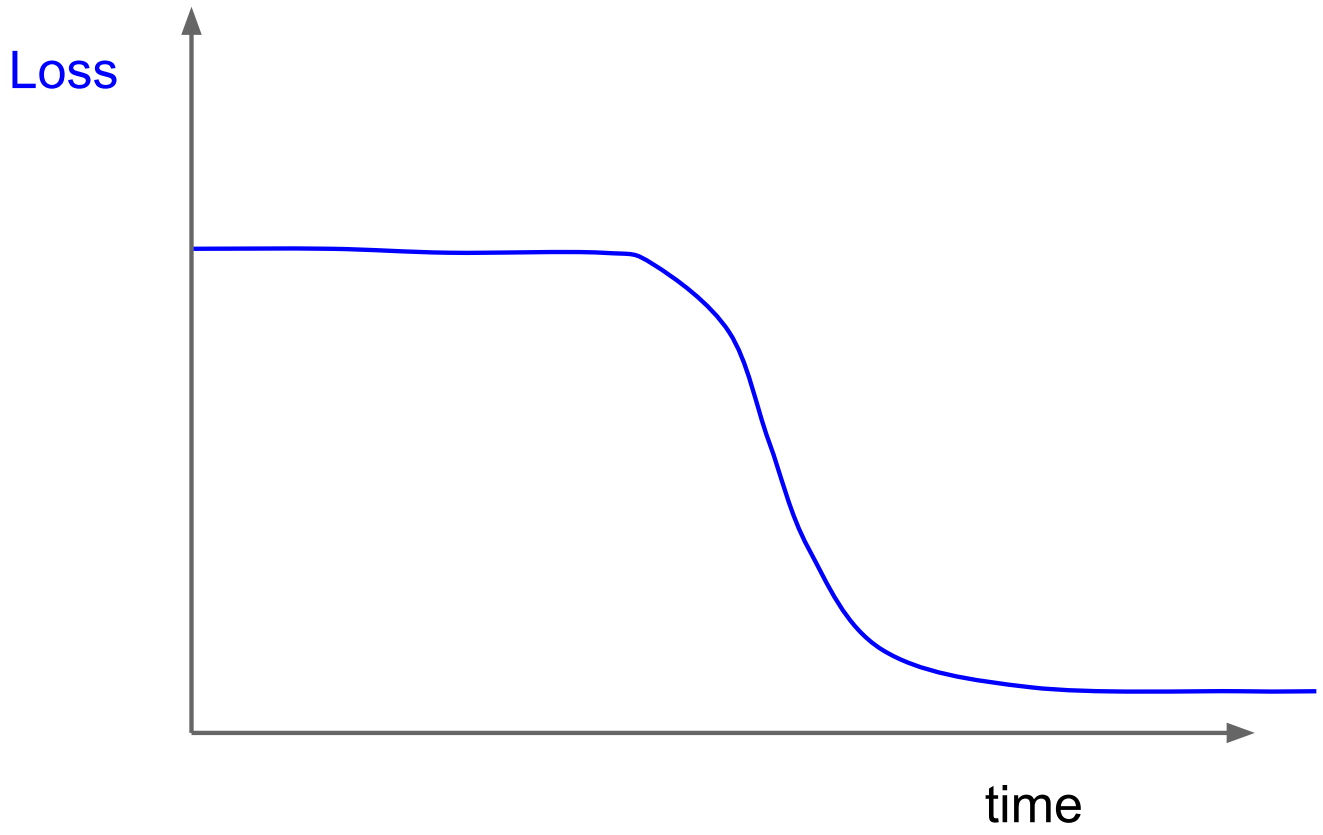



Evolution of your training loss is telling you something!

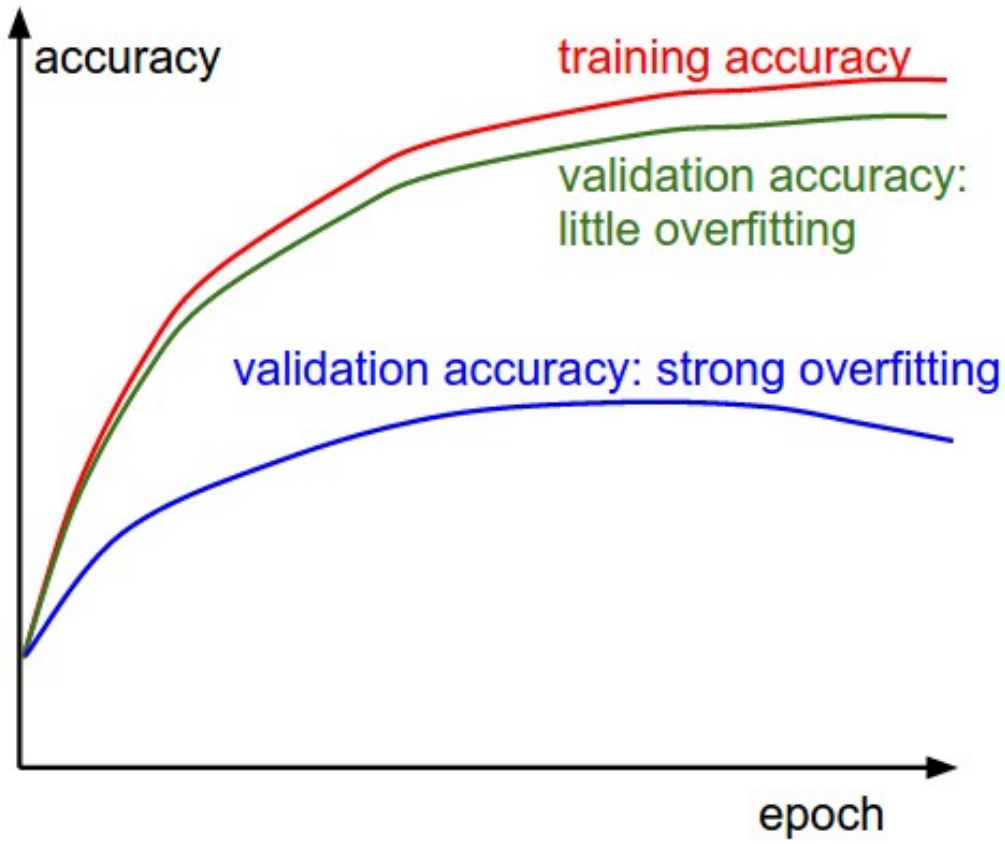



Typical training loss over time

# Telltale sign of a bad initialization



# Monitor & visualize the accuracy

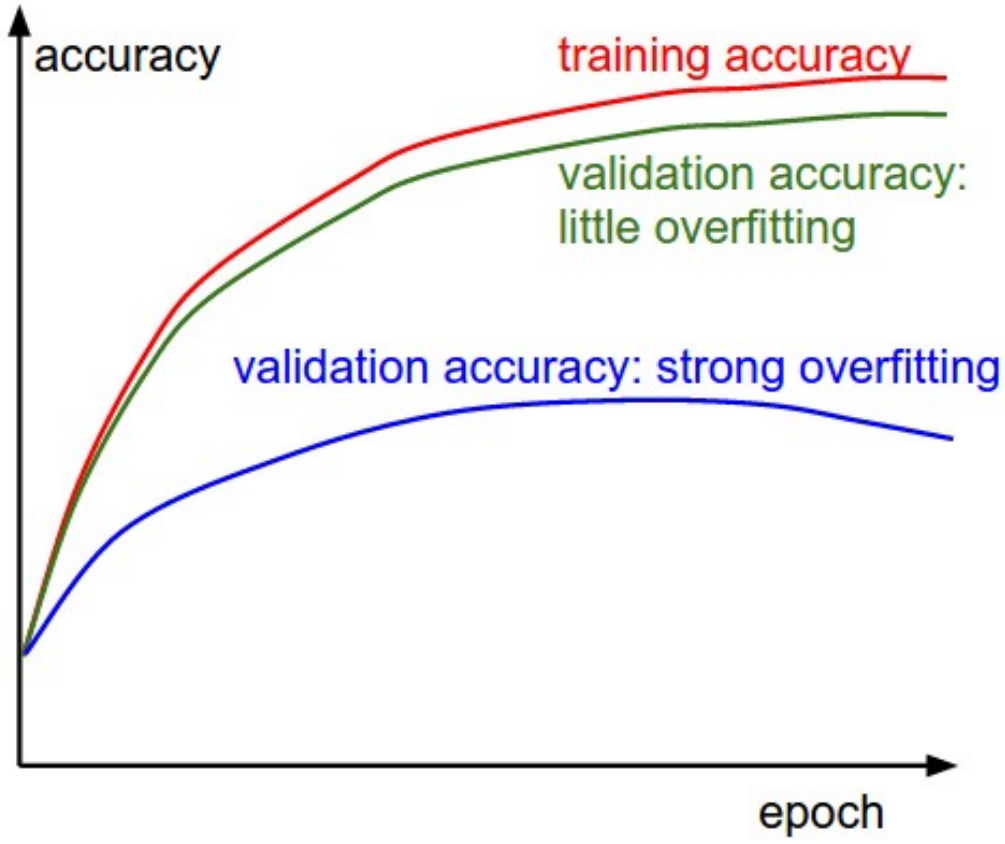



Gap between training and validation accuracy indicates amount of over-fitting.

Over-fitting $\Longrightarrow$ should increase regularization during training:

- increase the degree of $L_2$ regularization
- more dropout
- use more training data.

# Monitor & visualize the accuracy




Gap between training and validation accuracy indicates amount of over-fitting.

Under-fitting $\Longrightarrow$ model capacity not high enough:

- increase the size of the network

# Track the ratio of weight updates to weight magnitudes

- Track the **ratio** of the magnitude of the update vector to the magnitude of the parameter vector.
- So for a weight matrix, $W$, and vanilla SGD updates:

$$r = \frac{\| -\eta \nabla_W J \|}{\|W\|}$$

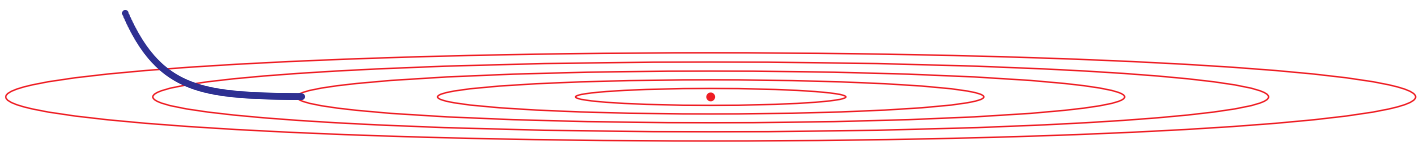
- A rough heuristic is that $r \sim .001$.
- If $r \ll .001 \implies$ learning rate might be too low.
- If $r \gg .001 \implies$ learning rate might be too high.

# Parameter Updates: Variations of Stochastic Gradient Descent

# One weakness of SGD

- SGD can be very slow.....
- **Example**: Use SGD to find the optimum of

$$f(\mathbf{x}) = -\exp(-.5\mathbf{x}^T\Sigma\mathbf{x})$$



$150$ iterations, $\eta = .01$

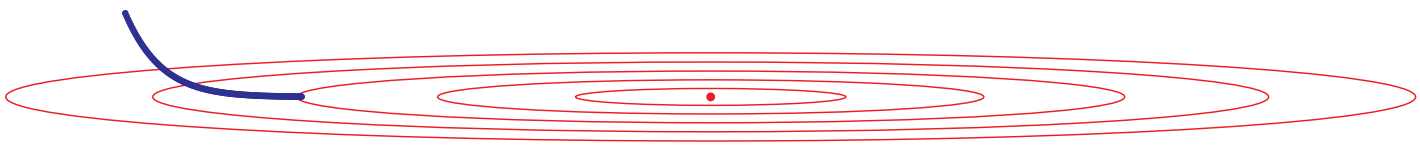
Curves show the iso-contours of $f(\mathbf{x})$

- Speed up optimization by increasing the learning rate?

# One weakness of SGD

- SGD can be very slow.....
- **Example**: Use SGD to find the optimum of

$$f(\mathbf{x}) = -\exp(-.5\mathbf{x}^T\Sigma\mathbf{x})$$

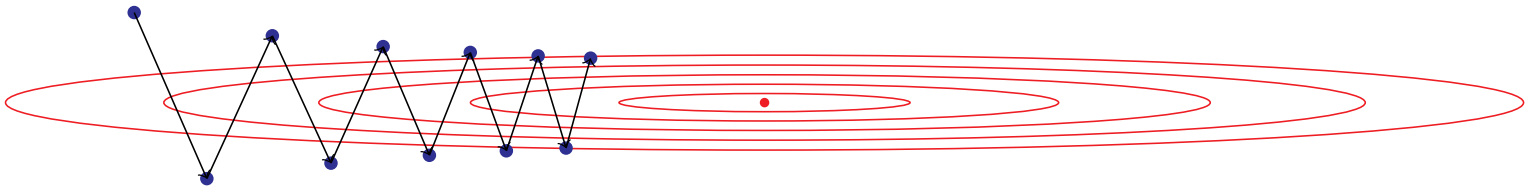


150 iterations, $\eta = .01$

Curves show the iso-contours of $f(\mathbf{x})$

- Speed up optimization by increasing the learning rate?

# One weakness of SGD



$10$ iterations, $\eta = .5$

(ridiculous learning rate to visualize effect)

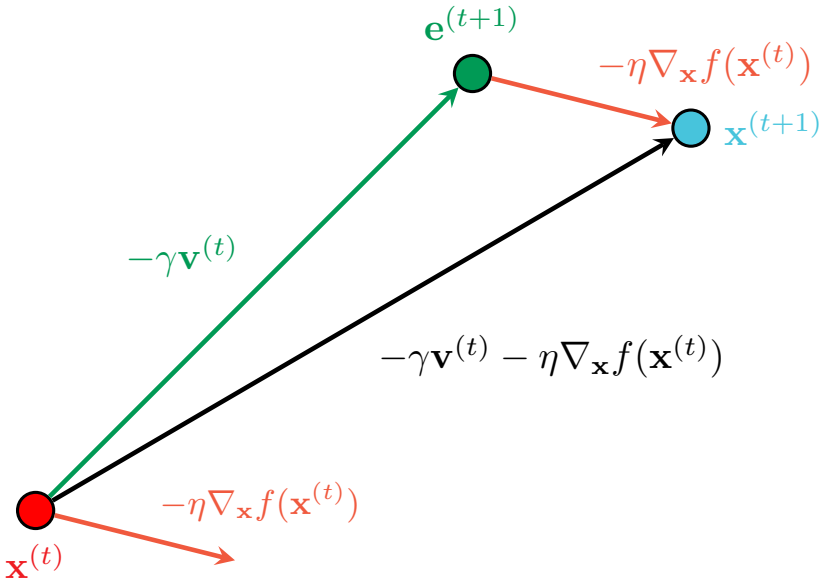
- SGD has trouble navigating ravines with high learning rates
  - SGD oscillates across the slopes of the ravine.
  - Makes slow progress along the bottom towards the local optimum.
- Unfortunately, ravines are common around local optima.

# Solution: SGD with momentum

- Introduce **momentum** vector as well as the gradient vector.
- Let $\gamma \in [0, 1]$ and $\mathbf{v}$ is the momentum vector

$$\mathbf{v}^{(t+1)} = \gamma\, \mathbf{v}^{(t)} + \eta\, \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)}) \quad \leftarrow \text{update vector}$$
$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \mathbf{v}^{(t+1)}$$

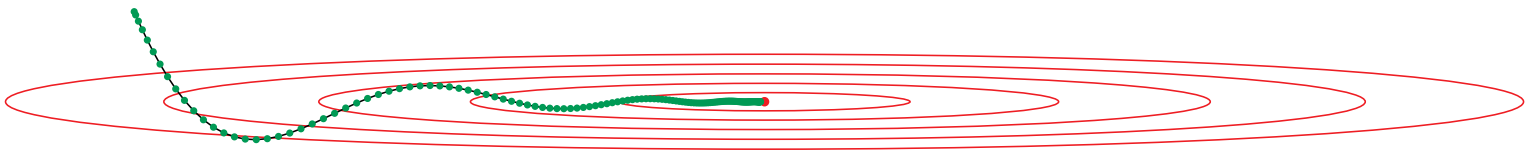
Typically $\gamma$ set to .9.



# How and why momentum helps

How?

- Momentum helps accelerate SGD in the appropriate direction.
- Momentum dampens the oscillations of default SGD.
  $\implies$ **Faster convergence**.



($\gamma = .9$, $\eta = .01$, 150 iterations)

Why?

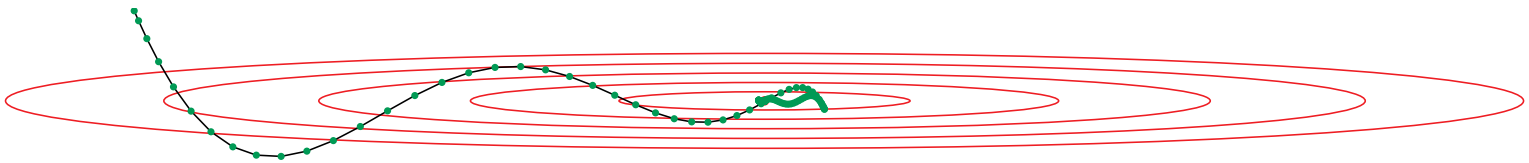
- For dimensions whose gradient is constantly changing then their entries in the update vector are damped.
- For dimensions whose gradient is **approx. constant** then their entries in the update vector are **not damped**.

# Momentum not the complete answer

- When using momentum

  $\Longrightarrow$ can pick up too much speed in one direction.

  $\Longrightarrow$ can overshoot the local optimum.



$(\gamma = .9, \eta = .03)$

# Solution: Nesterov accelerated gradient (NAG)

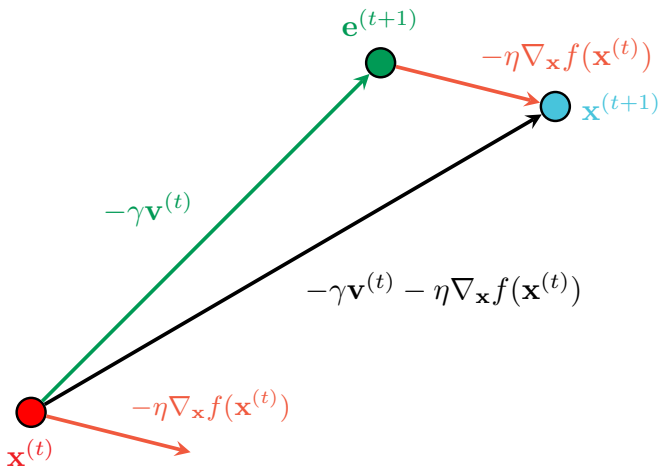
- Look and measure ahead.
- Use gradient at an estimate of the parameters at the next iteration.
- Let $\gamma \in [0, 1]$ then

$$\mathbf{e}^{(t+1)} = \mathbf{x}^{(t)} - \gamma \mathbf{v}^{(t)} \quad \leftarrow \text{estimate of } \mathbf{x}^{(t+1)}$$

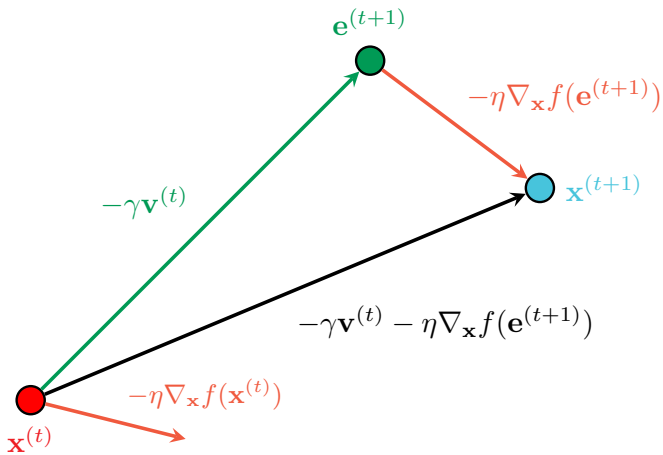
$$\mathbf{v}^{(t+1)} = \gamma\, \mathbf{v}^{(t)} + \eta\, \nabla_{\mathbf{x}} f(\mathbf{e}^{(t+1)}) \quad \leftarrow \text{update vector}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \mathbf{v}^{(t+1)}$$

Typically $\gamma$ set to .9.



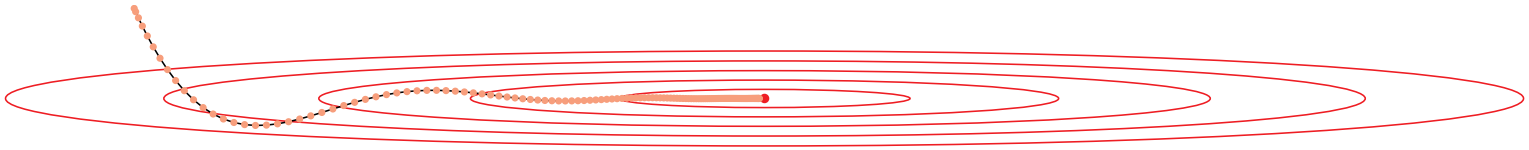

**Momentum update**




**NAG update**

# How and why NAG helps

- The anticipatory update prevents the algorithm having too large updates and overshooting.
- Algorithm has increased responsiveness to the landscape of $f$.



($\gamma = .9$, $\eta = .01$, 150 iterations)

**Note**:
NAG shown to greatly increase the ability to train RNNs:

Bengio, Y., Boulanger-Lewandowski, N. & Pascanu, R. *Advances in Optimizing Recurrent Networks, (2012).*

`http://arxiv.org/abs/1212.0901`

# More convenient form of NAG update

- Let $\gamma \in [0, 1]$ then

$$\mathbf{e}^{(t+1)} = \mathbf{x}^{(t)} - \gamma \mathbf{v}^{(t)} \quad \leftarrow \text{estimate of } \mathbf{x}^{(t+1)}$$

$$\mathbf{v}^{(t+1)} = \gamma\, \mathbf{v}^{(t)} + \eta\, \nabla_{\mathbf{x}} f(\mathbf{e}^{(t+1)}) \quad \leftarrow \text{update vector}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \mathbf{v}^{(t+1)}$$

- Form of update inconvenient as usually have $\mathbf{x}^{(t)}, \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)})$.
  - Can make a variable transformation

$$\mathbf{m}^{(t)} = \mathbf{x}^{(t)} - \gamma \mathbf{v}^{(t)} \quad \left(\equiv \mathbf{e}^{(t+1)}\right)$$

  - Can write $\mathbf{x}^{(t+1)}$ as

$$\mathbf{x}^{(t+1)} = \mathbf{m}^{(t+1)} + \gamma \mathbf{v}^{(t+1)} = \mathbf{x}^{(t)} - \mathbf{v}^{(t+1)}$$

$$\Longrightarrow$$

$$\begin{aligned} \mathbf{m}^{(t+1)} &= \mathbf{x}^{(t)} - (1+\gamma)\mathbf{v}^{(t+1)} \\ &= \mathbf{m}^{(t)} + \gamma \mathbf{v}^{(t)} - (1+\gamma)\mathbf{v}^{(t+1)} \end{aligned}$$

where

$$\mathbf{v}^{(t+1)} = \gamma\, \mathbf{v}^{(t)} + \eta\, \nabla_{\mathbf{x}} f(\mathbf{m}^{(t)})$$

# Improvements to NAG

- Want to adapt the updates to each individual parameter.
- Perform larger or smaller updates depending on the landscape of the cost function.
- Family of algorithms with **adaptive learning rates**
  - AdaGrad
  - AdaDelta
  - RMSProp
  - Adam

# AdaGrad

- Fora cleaner statement introduce some notation:

$$\mathbf{g}_t = \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)}) \quad \text{and } \mathbf{g}_t = (g_{t,1}, \ldots, g_{t,d})^T.$$

- Keep a record of the sum of the squares of the gradients w.r.t. each $x_i$ up to time $t$:

$$G_{t,i} = \sum_{j=1}^{t} g_{j,i}^2$$

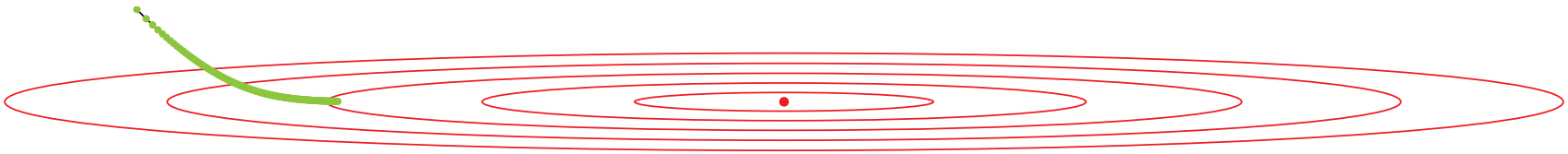
- The AdaGrad update step for each dimension is

$$x_i^{(t+1)} = x_i^{(t)} - \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

- Usually set $\epsilon = 1e-8$ and $\eta = .01$.

J. Duchi, E. Hazan & Y. Singer, *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research, 2011.

# Adagrad's convergence on our toy problem



($\epsilon = 1e-8$, $\eta = .01$, 150 iterations)

# Big weakness of AdaGrad

- Each $g_{t,i}^2$ is positive.

  $\Longrightarrow$ Each $G_{t,i} = \sum_{j=1}^{t} g_{j,i}^2$ keeps growing during training.

  $\Longrightarrow$ the effective learning rate $\eta/(\sqrt{G_{t,i} + \epsilon})$ shrinks and eventually $\longrightarrow 0$.

  $\Longrightarrow$ updates of $\mathbf{x}^{(t)}$ stop.

# AdaDelta

- Devised as an improvement to AdaGrad.
- Tackles AdaGrad's convergence to zero of the learning rate as $t$ increases.
- AdaDelta's two central ideas
  - scale learning rate based on the previous gradient values (like AdaGrad) but only using a recent time window,
  - include an acceleration term (like momentum) by accumulating prior updates.

M. Zeiler, *ADADELTA: An Adaptive Learning Rate Method*, 2012. `http://arxiv.org/abs/1212.5701`

# Technical details of AdaDelta

- Compute gradient vector $\mathbf{g}_t$ at current estimate $\mathbf{x}^{(t)}$.
- Update average of previous squared gradients (AdaGrad-like step)

$$\tilde{G}_{t,i} = \rho\, \tilde{G}_{t-1,i} + (1-\rho)\, g_{t,i}^2$$

- Compute the update vector

$$u_{t,i} = \frac{\sqrt{U_{t-1,i} + \epsilon}}{\sqrt{\tilde{G}_{t,i} + \epsilon}}\, g_{t,i}$$

- Compute exponentially decaying average of updates (momentum-like step)

$$U_{t,i} = \rho\, U_{t-1,i} + (1-\rho)\, u_{t,i}^2$$

- The AdaDelta update step:

$$x_i^{(t+1)} = x_i^{(t)} - u_{t,i}$$

# RMSProp

Also addresses AdaGrad's radically diminishing learning rate:

- RMSProp an **adaptive learning rate** method proposed by Geoff Hinton in *Lecture 6e of his Coursera Class*.
- Stores an exponentially decaying average of the square of the gradient vector:

$$E\left[\mathbf{g}_{t+1}^2\right] = \gamma\ E\left[\mathbf{g}_t^2\right] + (1-\gamma)\,\mathbf{g}_{t+1}^2$$

- The RMSProp update rule:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \frac{\eta}{\sqrt{E\left[\mathbf{g}_{t+1}^2\right] + \epsilon}}\,\mathbf{g}_{t+1}$$

- Typically set $\gamma = .9$ and $\eta = 0.001$.

# Adaptive Moment Estimation (Adam)

- Computes **adaptive learning rates** for each parameter.
- How?
  - Stores an **exponentially decaying average** of
    - ⋆ past gradients $\mathbf{m}^{(t)}$ and
    - ⋆ past squared gradients $\mathbf{v}^{(t)}$
  - $\mathbf{m}^{(t)}$ and $\mathbf{v}^{(t)}$ estimate the mean and variance of the sequence of computed gradients in each dimension.
  - Uses the variance estimate to
    - ⋆ damp the update in dimensions varying alot and
    - ⋆ increase the update in dimensions with low variation.

D. P. Kingma & J. L. Ba, *Adam: a Method for Stochastic Optimization*, International Conference on Learning Representations, 2015.

# Update equations for Adam

- Let $\mathbf{g}_t = \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)})$

$$\mathbf{m}^{(t+1)} = \beta_1 \mathbf{m}^{(t)} + (1 - \beta_1)\, \mathbf{g}_t$$
$$\mathbf{v}^{(t+1)} = \beta_2 \mathbf{v}^{(t)} + (1 - \beta_2)\, \mathbf{g}_t \;.*\; \mathbf{g}_t$$

- Set $\mathbf{m}^{(0)} = \mathbf{v}^{(0)} = \mathbf{0} \implies \mathbf{m}^{(t)}$ and $\mathbf{v}^{(t)}$ are biased towards zero (especially during the initial time-steps).
- Counter these biases by setting:

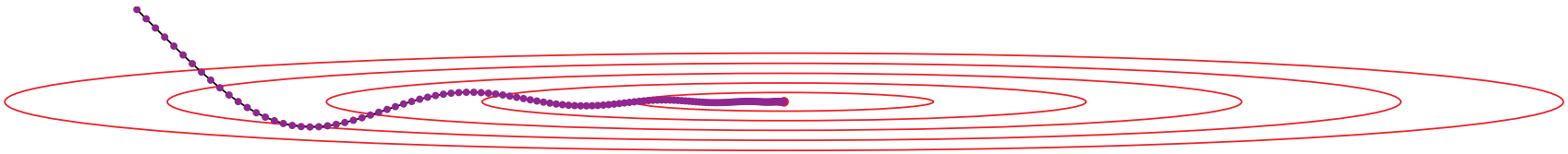
$$\hat{\mathbf{m}}^{(t+1)} = \frac{\mathbf{m}^{(t+1)}}{1 - \beta_1^t}, \qquad \hat{\mathbf{v}}^{(t+1)} = \frac{\mathbf{v}^{(t+1)}}{1 - \beta_2^t}$$

- The Adam update rule:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(t+1)}} + \epsilon}\, \hat{\mathbf{m}}^{(t+1)}$$

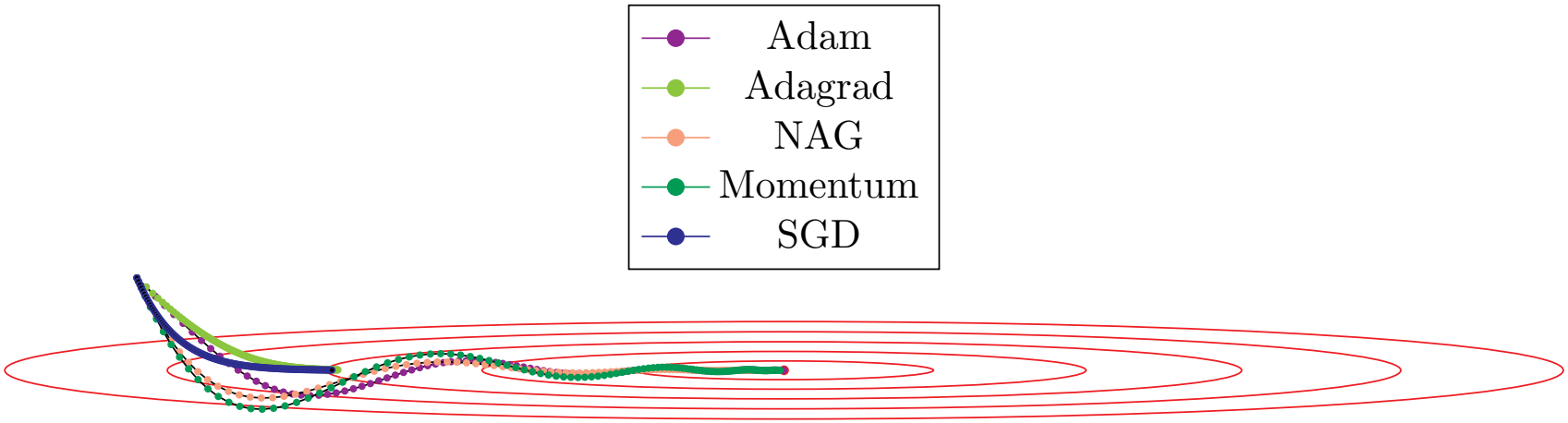
- Suggested default values $\beta_1 = .9, \beta_2 = .999, \epsilon = 10^{-8}$.

# Adam's performance on our toy problem

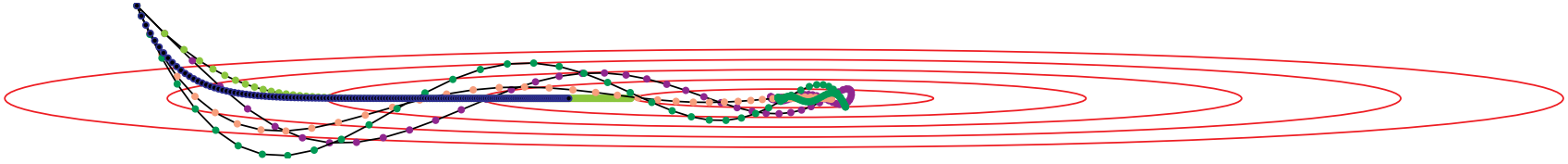


(default parameter settings, 150 iterations)

# Comparison of different algorithms on our toy problem

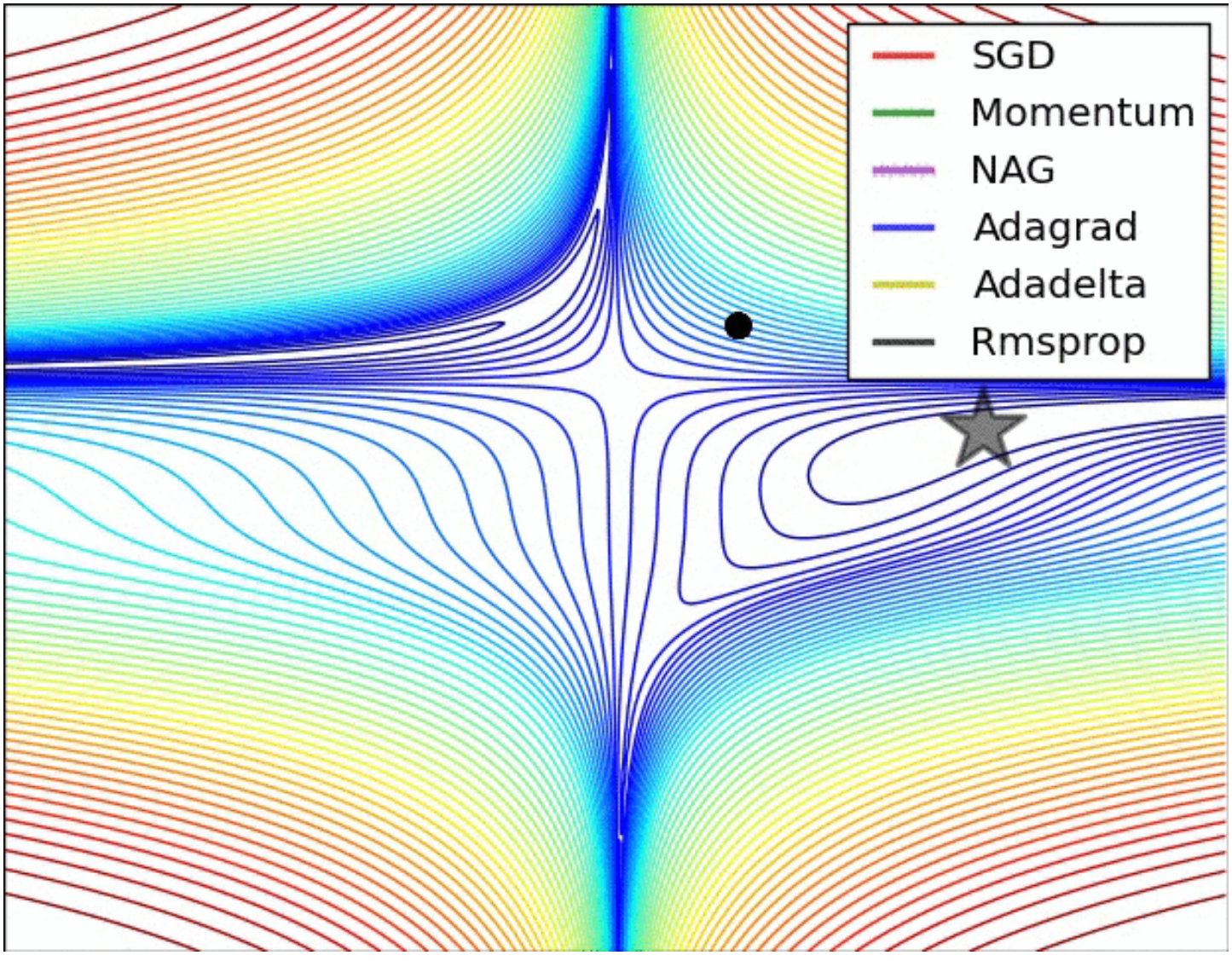



($\epsilon = 1e-8$, $\gamma = .9$, $\eta = .01$, 150 iterations)

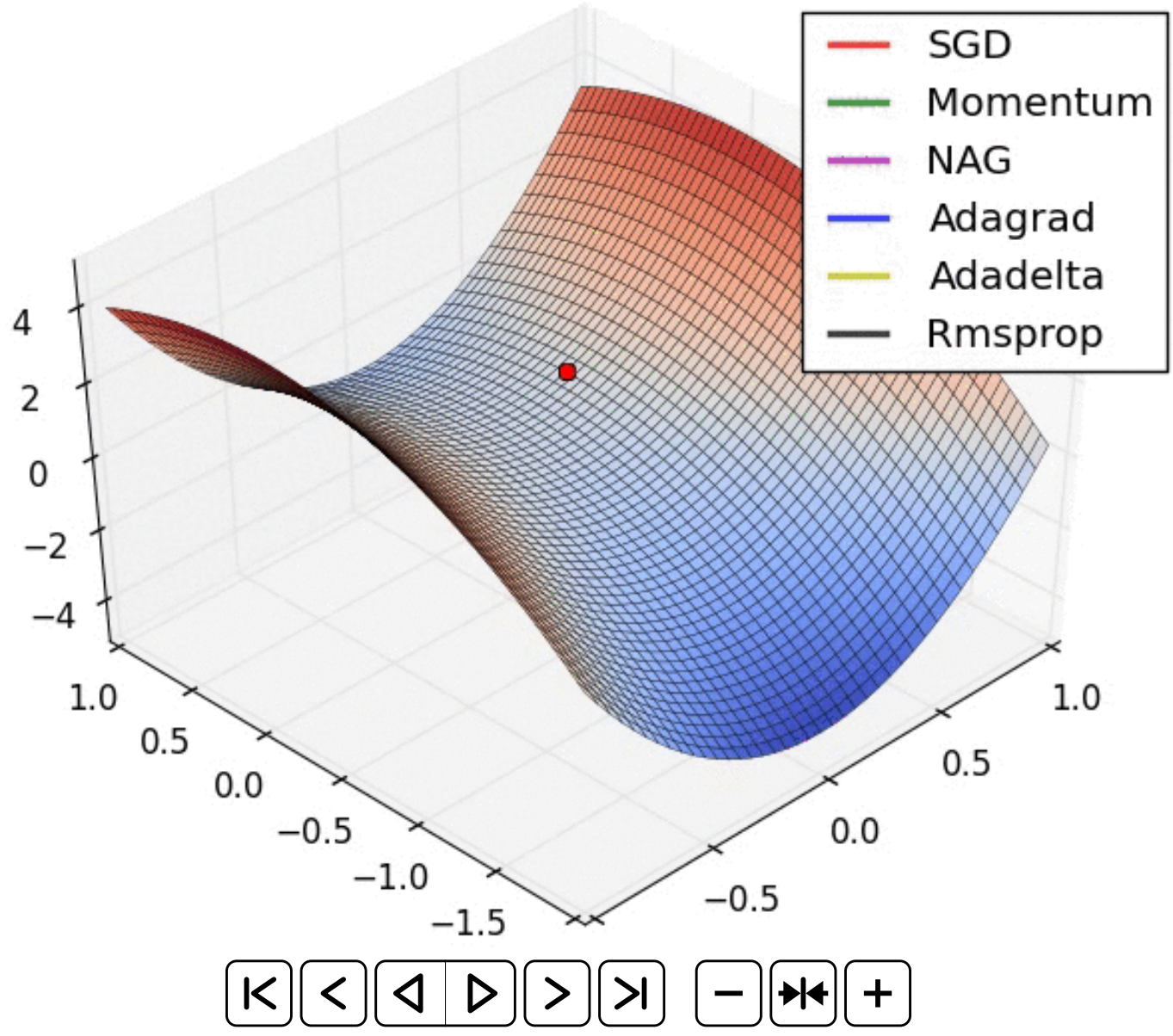


($\epsilon = 1e-8$, $\gamma = .9$, $\eta = .03$, 150 iterations)

# Comparison of different algorithms



# Comparison of different algorithms at a saddle point



# Which optimizer to use?

- Data sparse $\implies$ likely to achieve best results using one of the adaptive learning-rate methods.
- Using the adaptive learning-rate methods $\implies$ won't need to tune the learning rate (much!).
- RMSprop, AdaDelta, and Adam are very similar algorithms that do well in similar circumstances.
- Adam slightly outperforms RMSProp near the end of optimization.
- Adam might be the best overall choice.
- But vanilla SGD (without momentum) and a simple learning rate annealing schedule may be sufficient. But time until finding a local minimum may be long....

Annealing the learning rate

# Useful to anneal the learning rate

- When training deep networks, usually helpful to anneal the learning rate over time.
- **Why?**
  - Stops the parameter vector from bouncing around too widely.
  - $\Longrightarrow$ can reach into deeper, but narrower parts of the loss function.
- But knowing when to decay the learning rate is tricky!
- Decay too slowly $\Longrightarrow$ waste computations bouncing around chaotically with little improvement.
- Decay too aggressively $\Longrightarrow$ system unable to reach the best position it can.

# Common approaches to learning rate decay

- **Step decay**:
  After every $n$th epoch set

$$\eta = \alpha\eta$$

  where $\alpha \in (0, 1)$. (Instead sometimes people monitor the validation loss and reduce the learning rate when this loss stops improving.)

- **Exponential decay**:

$$\eta = \eta_0 e^{-kt}$$

  where $t$ is iteration number (either w.r.t. number of update steps or epochs). Then $\eta_0$ and $k$ are hyper-parameters.

- $1/t$ **decay**:

$$\eta = \frac{\eta_0}{1 + kt}$$

# Common approaches to learning rate decay

- **Step decay**:
  After every $n$th epoch set

$$\eta = \alpha\eta$$

  where $\alpha \in (0, 1)$. (Instead sometimes people monitor the validation loss and reduce the learning rate when this loss stops improving.)

- **Exponential decay**:

$$\eta = \eta_0 e^{-kt}$$

  where $t$ is iteration number (either w.r.t. number of update steps or epochs). Then $\eta_0$ and $k$ are hyper-parameters.

- $1/t$ **decay**:

$$\eta = \frac{\eta_0}{1 + kt}$$

Step decay most common. Better to decay conservatively and train for longer.

Optimization of the training hyper-parameters

# Hyperparameters to adjust

- Initial learning rate.
- Learning rate decay schedule.
- Regularization strength
  - $L_2$ penalty
  - Dropout strength

# Cross-validation strategy

- Do a **coarse** $\rightarrow$ **fine** cross-validation in stages.
- **Stage 0**: Identify the range of feasible learning rates & regularization penalties. (usually done interactively and train only for a few updates.)
- **Stage 1**: Broad search. Goal is to narrow the search range. Only run training for a few epochs.
- **Stage 2**: Finer search. Increase training times.
- **Stage ...**: Repeat Stage 2 as necessary.

Use performance on the **validation set** to identify good hyper-parameter settings.

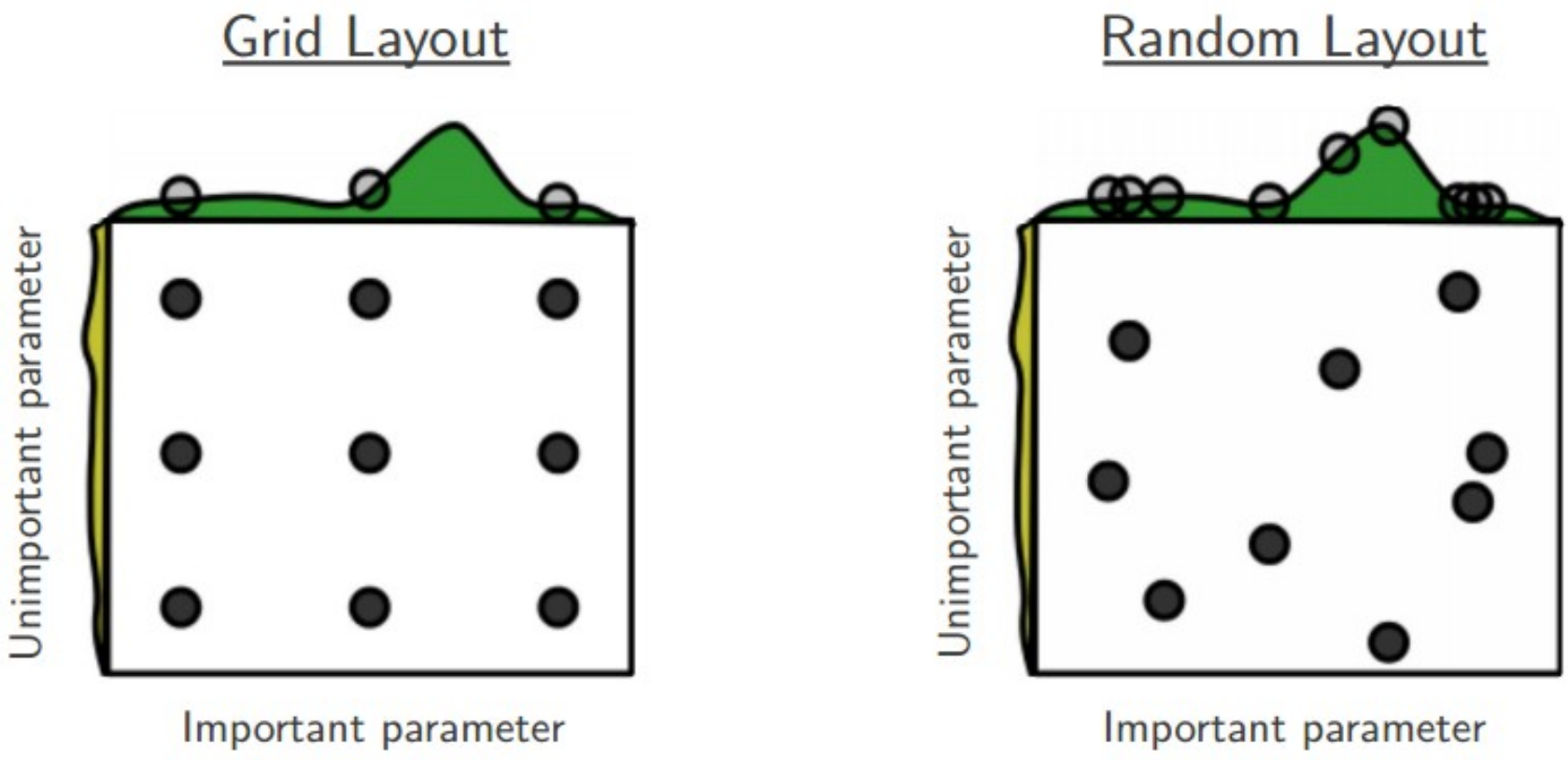
# Hyper-parameter ranges

- Search for the **learning-rate** and **regularization** hyperparameters on a log scale.

  **Example**:
  Generate a potential learning rate with

$$\alpha = \text{uniform}(-6, 1)$$
$$\eta = 10^{\alpha}$$

# Prefer random search to grid search




"*randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid*"

**Random Search for Hyper-Parameter Optimization**, Bergstra and Bengio, 2012

# Evaluation: Model Ensembles

# Model Ensembles

- Train multiple independent models (same hyper-parameter settings, different initializations). ($\sim$ 5 models)
- At test time apply each model and average their results.

# Model Ensemble on the cheap

- Can also get a small boost from averaging multiple model checkpoints of a single model.
- At test time apply each model and average their results.

# Estimating Test Error

(just so that everyone knows what is acceptable and what's not)

# Measuring the performance of a classifier

- Have learnt a classification $f(\cdot \mid \hat{\boldsymbol{\theta}})$ from the training data $\mathcal{D}$
- How well does $f(\cdot \mid \hat{\boldsymbol{\theta}})$ **generalize** to unseen examples?
- Does $f(\mathbf{x}_{\text{new}} \mid \hat{\boldsymbol{\theta}}) = y_{\text{new}}$ for a large number of $(\mathbf{x}_{\text{new}}, y_{\text{new}})$?

# Estimating the Generalization Ability

- To measure accuracy of $f(\cdot \mid \hat{\boldsymbol{\theta}})$ ideally would compute the **Expected loss**:

$$\mathrm{E}\left[l(Y, f(\mathbf{X} \mid \hat{\boldsymbol{\theta}})\right] = \int_{\mathbf{x}} \int_{y} l\left(y, f(\mathbf{x} \mid \hat{\boldsymbol{\theta}})\right) \, p_{\mathbf{X},Y}(\mathbf{x}, y) \, d\mathbf{x} \, dy$$

  where

  - $l(y, f(\mathbf{x} \mid \hat{\boldsymbol{\theta}}))$ measures how well $f(\mathbf{x} \mid \hat{\boldsymbol{\theta}})$ predicts labels $y$.

# Estimating the Error rate

- Usually don't know the distribution $p_{\mathbf{X},Y}(\mathbf{x}, y)$.

  $\Rightarrow$ cannot compute the Expected loss

- Instead maybe one could consider the **Training Error**:

$$\frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},y)\in\mathcal{D}} l\left(y, f(\mathbf{x} \mid \hat{\boldsymbol{\theta}})\right)$$

- Training error frequently not a good proxy for the test performance.

- Especially if $\hat{\theta}$ has been estimated from $\mathcal{D}$ (and some parameter tuning has occurred).

- What is the standard thing to do then?

# Estimating the Error rate

- Usually don't know the distribution $p_{\mathbf{X},Y}(\mathbf{x}, y)$.

  $\Rightarrow$ cannot compute the Expected loss

- Instead maybe one could consider the **Training Error**:

$$\frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},y)\in\mathcal{D}} l\left(y, f(\mathbf{x} \mid \hat{\boldsymbol{\theta}})\right)$$

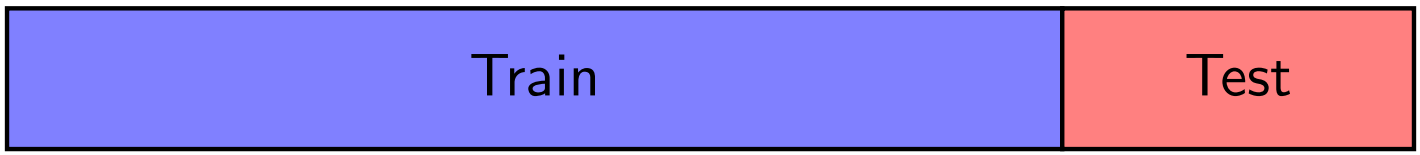
- Training error frequently not a good proxy for the test performance.

- Especially if $\hat{\boldsymbol{\theta}}$ has been estimated from $\mathcal{D}$ (and some parameter tuning has occurred).

- What is the standard thing to do then?

# For a data-rich situation

Randomly divide the dataset into 2 parts: $\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{test}}$.




Common split ratio 75%, 25%

- Use $\mathcal{D}_{\text{train}}$ to estimate $f$'s parameters $\hat{\boldsymbol{\theta}}$.
- Use $\mathcal{D}_{\text{test}}$ to compute the **test loss** for $f(\cdot \mid \hat{\boldsymbol{\theta}})$:

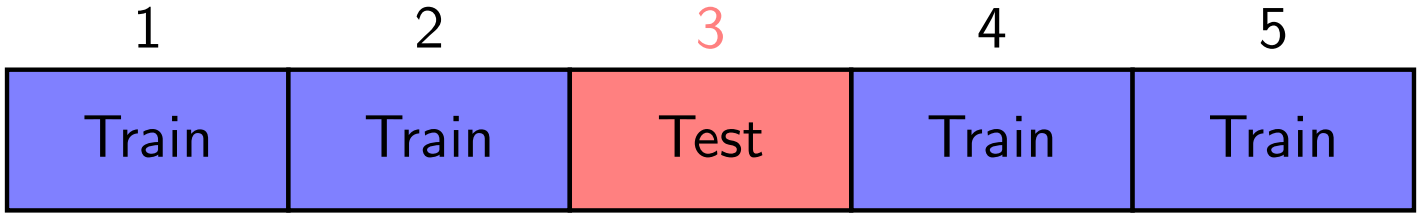
$$\text{Err} = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(\mathbf{x},y)\in\mathcal{D}_{\text{test}}} l\left(y, f(\mathbf{x} \mid \hat{\boldsymbol{\theta}})\right)$$

  an estimate of the expected loss.

However, if labelled data is scarce then your test set may be small and not so representative.

# When labelled data is scarce: $K$-Fold Cross-Validation

**General Approach**




- Partition the data into $K$ roughly equal-size subsets

$$\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_K$$

- Use $\mathcal{D}_k$ to estimate the test loss of $f$ where $\hat{\boldsymbol{\theta}}$ is calculated from $\mathcal{D} \setminus \mathcal{D}_k$.

- Do this for $k = 1, 2, \ldots, K$ and average the $K$ estimates of the expected loss. The is the cross validation (CV) error.

- The cross validation error $CV(f)$ is an estimate of the test loss.

# $K$-Fold Cross-validation: Detailed description

- The mapping $\kappa : \{1, \ldots, n\} \to \{1, \ldots, K\}$ indicates example $i$ belongs to partition $\kappa(i)$.
- Denote estimate of the parameters using data $\mathcal{D} \setminus \mathcal{D}_k$ by $\hat{\boldsymbol{\theta}}^{-k}$.
- **Cross-validation** estimate of the test loss is:

$$\mathrm{CV}(f) = \frac{1}{n} \sum_{i=1}^{n} l\left(y_i, f(x_i \mid \hat{\boldsymbol{\theta}}^{-\kappa(i)})\right)$$

- Typical choices for $K$ are 5 or 10.
- The case $K = n$ is known as leave-one-out cross-validation.

# Model/Classifier Selection

# Selecting between different classifiers

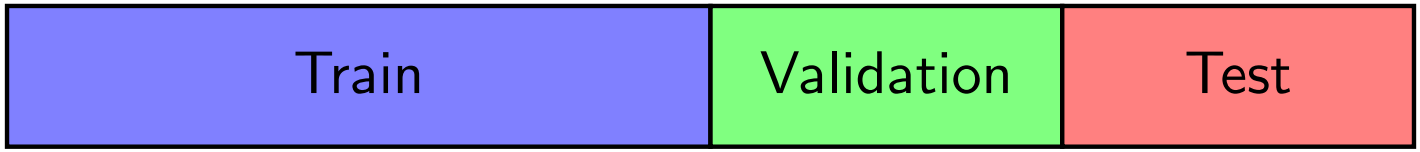
You can generate different classifiers $f_1, \ldots, f_m$ because you

1. Investigate different types of classifiers
   - Random forest,
   - Linear SVM,
   - Bayesian classifier, ...
2. Have same type of classifier but different architectures
   - Random forest but depth of trees differs,
   - Neural networks but number of nodes and layers differ,
   - Bayesian classifier with different class likelihoods, ...
3. Have same type of classifier but different tuning parameters
   - Linear SVMs but $C$ parameter differs,
   - Neural networks with different # of training iterations,
   - Kernel SVMs with different kernel parameters, ...
4. Any mixture of the above.

How do we choose the best classifier?

# For a data-rich situation

Randomly divide the dataset into 3 parts: $\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{val}} \cup \mathcal{D}_{\text{test}}$.




Common split ratio 50%, 25%, 25%.

## Model Selection

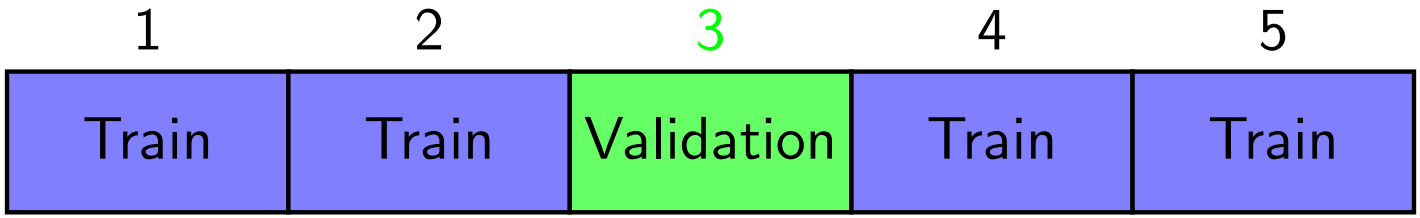
- Use **training set**, $\mathcal{D}_{\text{train}}$, to estimate $\hat{\boldsymbol{\theta}}_j$ for each $f_j$.
- Use **validation set**, $\mathcal{D}_{\text{val}}$, to estimate the test loss for each $f_j$.
- Choose $f_{j^*}$ as the $f_j$ with the lowest test loss estimate.

## Assessment of the chosen model

- Use $\mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{val}}$ to estimate $\hat{\boldsymbol{\theta}}_{j^*}$ for $f_{j^*}$.
- Use **test set** $\mathcal{D}_{\text{test}}$ - unseen till now - to estimate $f_{j^*}$'s test loss.

# When labelled data is scarce: $K$-Fold Cross-Validation

**General Approach**




- Partition the data into $K$ roughly equal-size subsets

$$\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_K$$

- Use the $\mathcal{D}_k$ to estimate the expected loss of $f_j$ where $\hat{\boldsymbol{\theta}}_j$ is calculated from $\mathcal{D} \setminus \mathcal{D}_k$.

- Do this for $k = 1, 2, \ldots, K$ and average the $K$ estimates of the expected loss. Compute the cross-validation error $CV(f_j)$ for each classifier.

- Select the classifier, $f_{j^*}$, with lowest cross validation error.

# **Cross-Validation** for
# Model Selection & Model Assessment

# Option 1

- For each classifier $f_j$ compute its $K$-fold cross-validation error $CV(f_j)$
- Choose the classifier $f_{j^*}$ such that

$$j^* = \underset{1 \leq j \leq m}{\arg\min} \; CV(f_j)$$

- The estimate of the test error of the best classifier is given by

$$CV(f_{j^*})$$

# Option 1

- For each classifier $f_j$ compute its $K$-fold cross-validation error $CV(f_j)$
- Choose the classifier $f_{j^*}$ such that

$$j^* = \underset{1 \leq j \leq m}{\arg\min} \; CV(f_j)$$

- The estimate of the test error of the best classifier is given by

$$CV(f_{j^*})$$

You may have some concerns.

Have used the same training data for

- model selection **and**
- model assessment.

$\Longrightarrow$ Chance you have over-estimated generalization ability of selected model.

# Option 2: Nested CV

**Measures the performance of your model selection process.**

1. Partition the data into $K_0$ folds $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \cdots \cup \mathcal{D}_{K_0}$.

2. for $k = 1, \ldots, K_0$
   - Set $\mathcal{E} = \mathcal{D} \setminus \mathcal{D}_k$
   - Perform $K_1$-fold cross-validation using $\mathcal{E}$ to select the best classifier $f_{j_k^*}$.
   - Compute the average loss of this classifier on $\mathcal{D}_k$

$$\text{Err}_k = \frac{1}{|\mathcal{D}_k|} \sum_{(\mathbf{x},y)\in\mathcal{D}_k} l\left(y, f_{j_k^*}(\mathbf{x}; \hat{\boldsymbol{\theta}}_{j_k^*})\right)$$

3. The cross-validation score for the model selection process is

$$CV(\text{model selection process}) = \frac{1}{K_0} \sum_{k=1}^{K_0} \text{Err}_k$$

And for deep neural network hyper-parameter tuning.

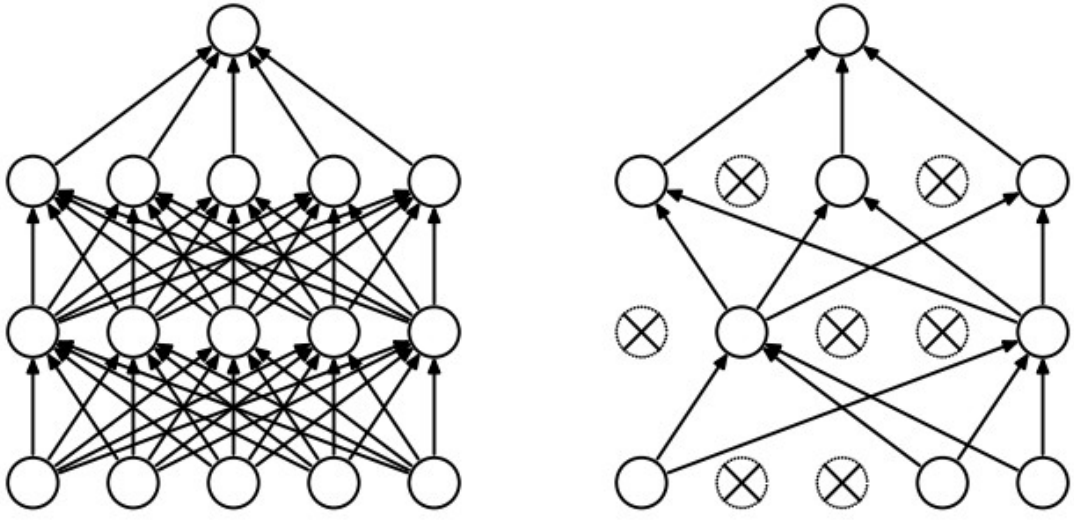
# One validation set

- Usually have one validation fold as opposed to cross-validation.
- Simplifies the code base
- Makes things computationally feasible.

Regularization Via **Dropout**

# Dropout

- **Randomly set some activations to zero in forward pass**



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

*[Srivastava et al.]*

- Training practice introduced by Hinton.

- **Note**: Each training sample in the mini-batch has its own random dropout mask.

# Training with Dropout

Set $p \in (0, 1]$ ← probability of keeping an activation active.

**The Forward Pass** (for a 2-layer network)

1. Compute the first set of activation values:

$$\mathbf{x}^{(1)} = \max(0, W_1 \mathbf{x}^{(0)} + \mathbf{b}_1)$$

2. Randomly choose which entries of $\mathbf{x}^{(1)}$ to switch off

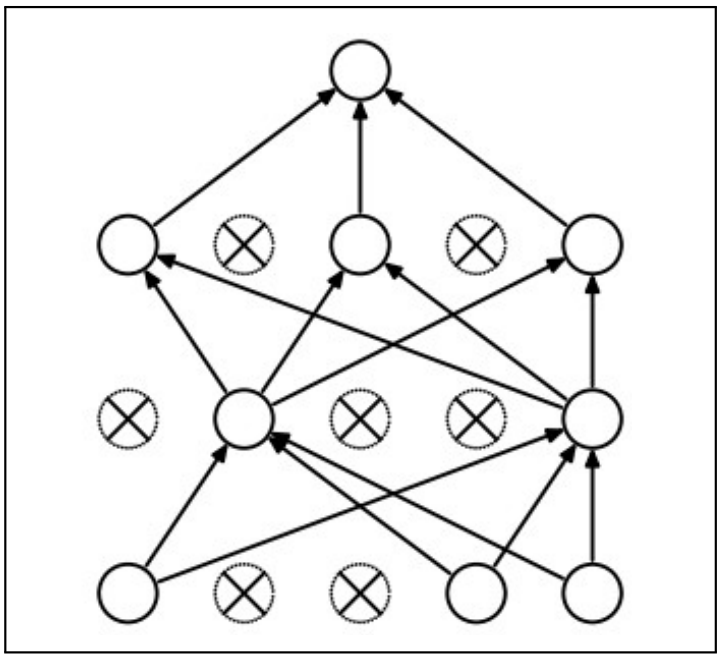
$$\mathbf{u}_1 = \text{rand}(\text{size}(\mathbf{x}^{(1)})) < p \quad \text{(Matlab notation)}$$
$$\mathbf{x}^{(1)} = \mathbf{x}^{(1)} .* \mathbf{u}_1$$

3. Repeat the process for the next layer

$$\mathbf{x}^{(2)} = \max(0, W_2 \mathbf{x}^{(1)} + \mathbf{b}_2)$$
$$\mathbf{u}_2 = \text{rand}(\text{size}(\mathbf{x}^{(2)})) < p \quad \text{(Matlab notation)}$$
$$\mathbf{x}^{(2)} = \mathbf{x}^{(2)} .* \mathbf{u}_2$$

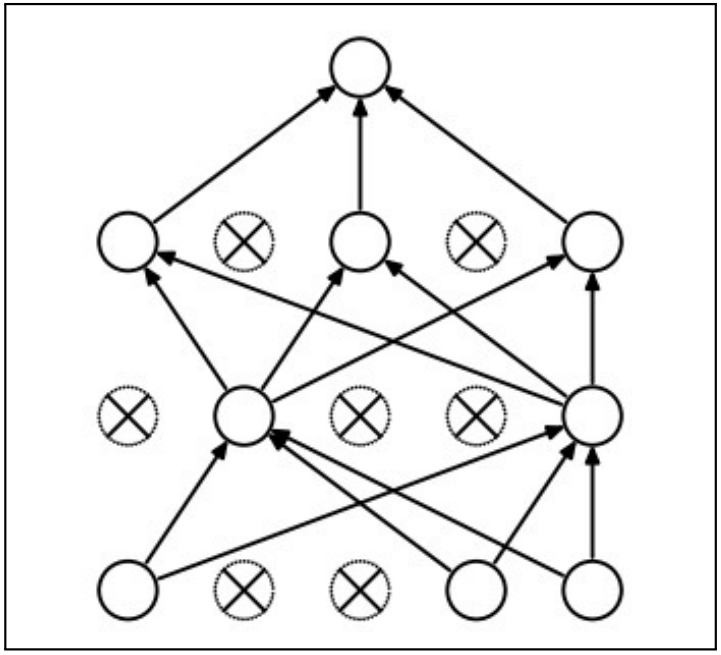
4. Output: $\mathbf{x}^{(3)} = \text{SoftMax}(W_3 \mathbf{x}^{(2)} + \mathbf{b}_3)$

# Why is this a good idea?



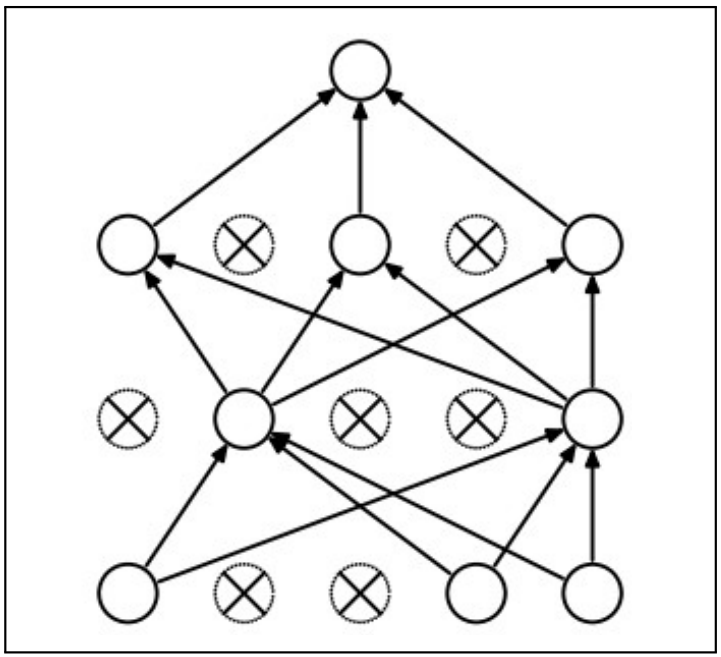
- Forces the network to have a redundant representation.
- **Another interpretation**
  - Dropout is training a large ensemble of models.
  - Each binary mask is one model, gets trained on only ∼one datapoint.

# Why is this a good idea?



- Forces the network to have a redundant representation.
- **Another interpretation**
  - Dropout is training a large ensemble of models.
  - Each binary mask is one model, gets trained on only $\sim$one datapoint.

# At test time

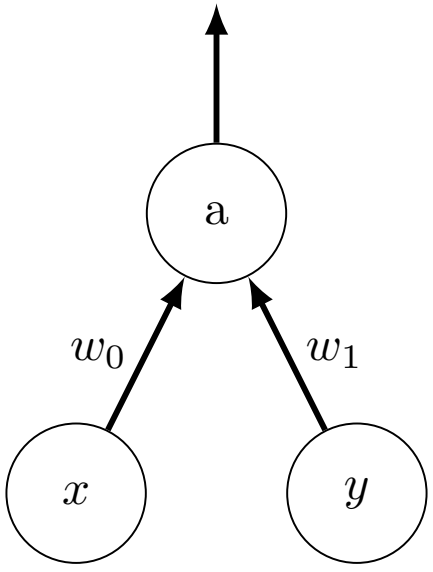


- **Ideally**: Want to integrate out all the noise.
- **Monte Carlo approximation**
  - Do many forward passes with different dropout masks.
  - Average all the predictions.

# At test time

- Can do this with a single forward pass (approximately).
- Leave all the activations turned on (no dropout).
- Surely we must compensate?

# At test time

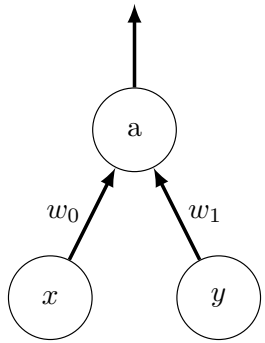



Consider this simple partial network

- During testing if we do not compensate:

$$a_{\text{test}} = w_0 x + w_1 y$$

# At test time



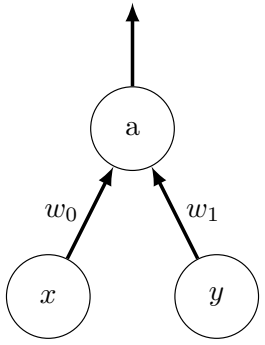

Consider this simple partial network

- During dropout training:
  1. Each input activation $x$ and $y$ is switched off with probability $1 - p$.
  2. The possible input activations are

| inputs | probability |
|---|---|
| $(0, 0)$ | $(1-p)^2$ |
| $(x, 0)$ | $p(1-p)$ |
| $(0, y)$ | $(1-p)p$ |
| $(x, y)$ | $p^2$ |

  3. $$\begin{aligned} \mathsf{E}[a_{\text{training}}] &= (1-p)^2(w_0 0 + w_1 0) + p(1-p)(w_0 x + w_1 0) \\ &\quad + p(1-p)(w_0 0 + w_1 y) + p^2(w_0 x + w_1 y) \\ &= p\,(w_0 x + w_1 y) \end{aligned}$$

# At test time




Consider this simple partial network

- During **testing** if we do not compensate:

$$a_{\text{test}} = w_0 x + w_1 y$$

- During **dropout training**:

$$\mathsf{E}[a_{\text{training}}] = p\,(w_0 x + w_1 y) = p\,a_{\text{test}}$$

$\Longrightarrow$ have to compensate at test time by scaling the activations by $p$.

# Must compensate at test time

- Must scale the activations so for each neuron:

  *output at test time = expected output at training time*

- Don't drop activations but have to compensate

$$\mathbf{x}^{(1)} = \max(0, W_1\mathbf{x}^{(0)} + \mathbf{b}_1) * p$$
$$\mathbf{x}^{(2)} = \max(0, W_2\mathbf{x}^{(1)} + \mathbf{b}_2) * p$$
$$\mathbf{x}^{(3)} = \mathrm{SoftMax}(W_3\mathbf{x}^{(2)} + \mathbf{b}_3)$$

# More common: Inverted Dropout

- During **training**:

$$\mathbf{x}^{(1)} = \max(0, W_1\mathbf{x}^{(0)} + \mathbf{b}_1)$$
$$\mathbf{u}_2 = (\text{rand}(\text{size}(\mathbf{x}^{(1)})) < p)/p \quad \longleftarrow \text{Note } /p$$
$$\mathbf{x}^{(1)} = \mathbf{x}^{(1)} .* \mathbf{u}_2$$

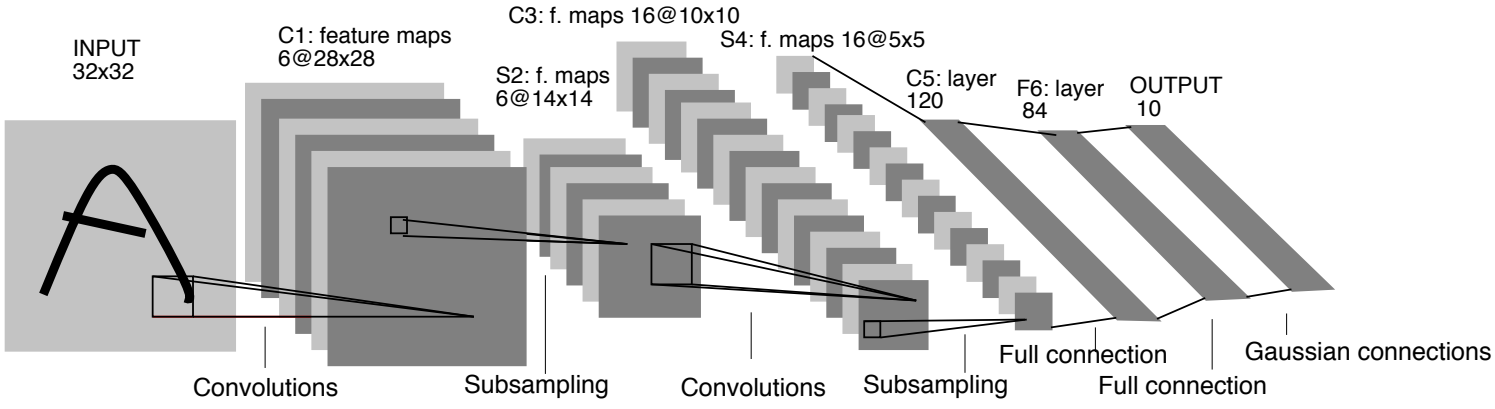
$$\mathbf{x}^{(2)} = \max(0, W_2\mathbf{x}^{(1)} + \mathbf{b}_2)$$
$$\mathbf{u}_2 = (\text{rand}(\text{size}(\mathbf{x}^{(2)})) < p)/p \quad \longleftarrow \text{Note } /p$$
$$\mathbf{x}^{(2)} = \mathbf{x}^{(2)} .* \mathbf{u}_3$$

$$\mathbf{x}^{(3)} = \text{SoftMax}(W_3\mathbf{x}^{(2)} + \mathbf{b}_3)$$

- $\Longrightarrow$ At **test time** no scaling necessary:

$$\mathbf{x}^{(1)} = \max(0, W_1\mathbf{x}^{(0)} + \mathbf{b}_1)$$
$$\mathbf{x}^{(2)} = \max(0, W_2\mathbf{x}^{(1)} + \mathbf{b}_2)$$
$$\mathbf{x}^{(3)} = \text{SoftMax}(W_3\mathbf{x}^{(2)} + \mathbf{b}_3)$$

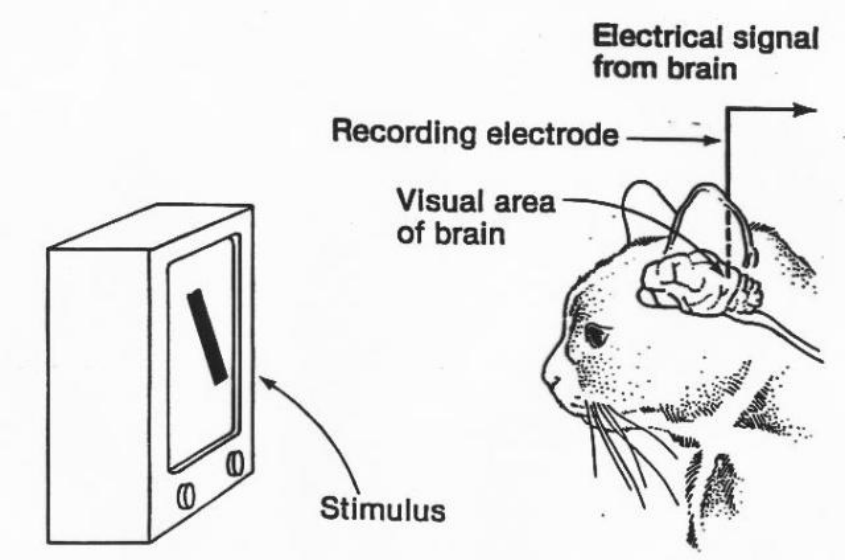
# Convolutional Neural Networks (ConvNets)

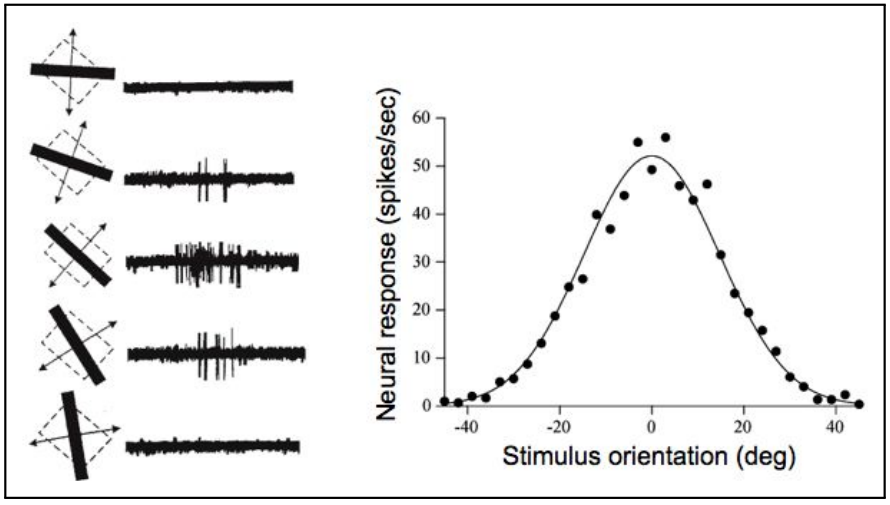



**LeNet-5** (LeCun '98)

# ConvNets: Some history

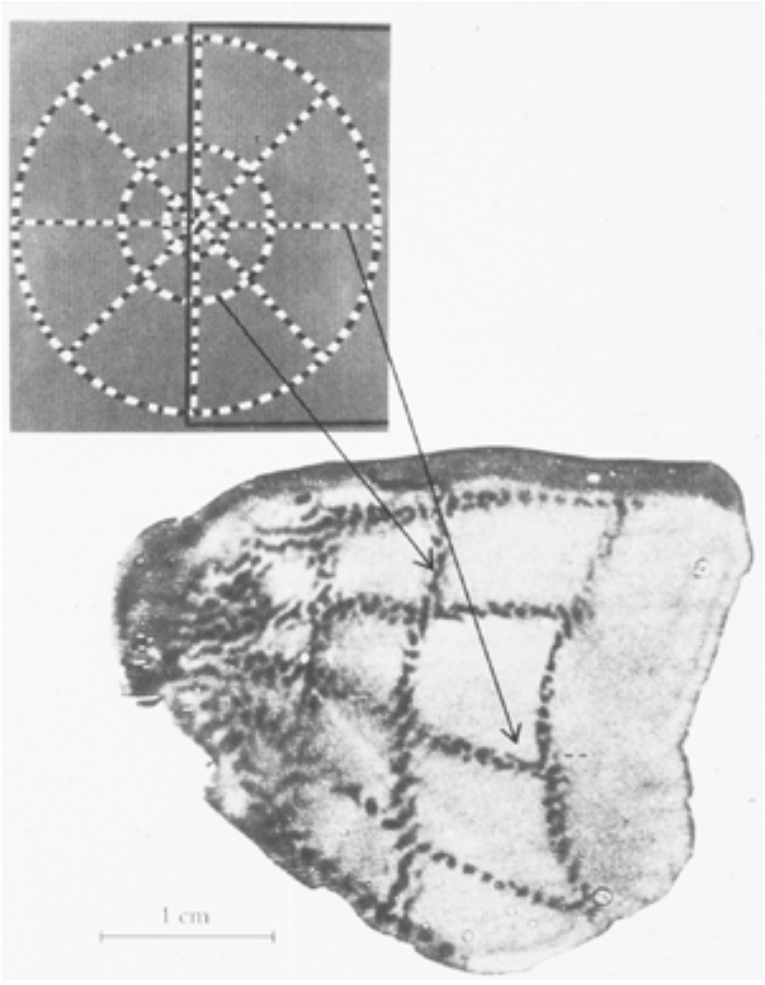
# Hubel & Wiesel cat experiments 1968



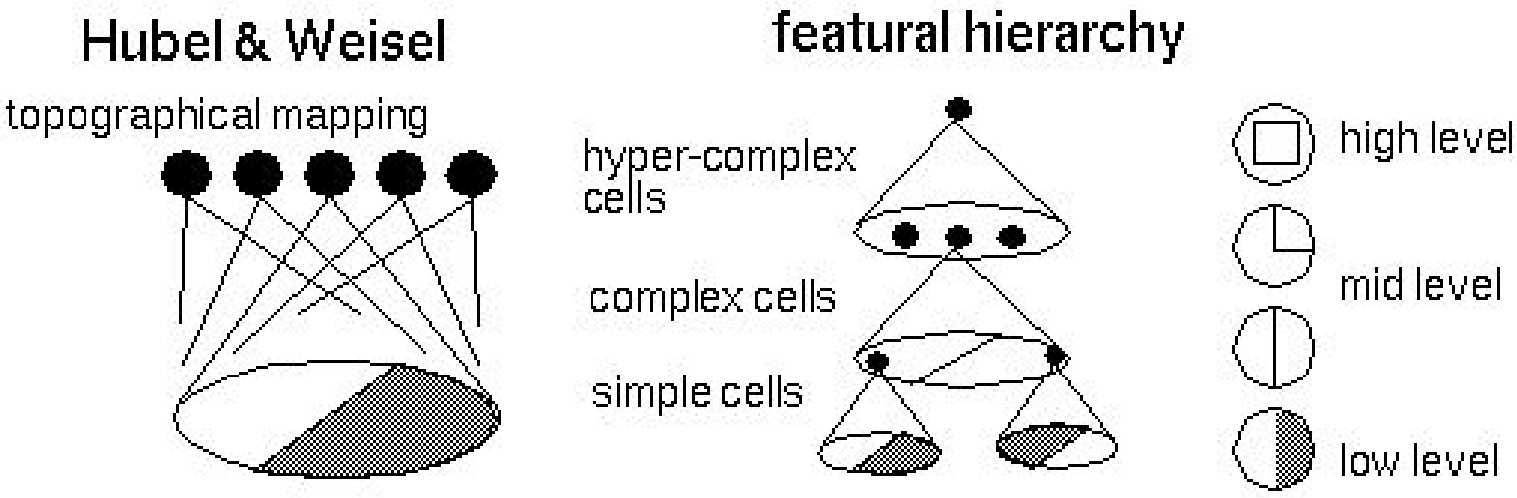

- Discovered *visual cortex* consists of a hierarchy of simple, complex, and hyper-complex cells. (Experiments in 50's & 60's)
- Hubel & Wiesel won the Nobel prize (1981).

# Hubel & Wiesel




**Topographical mapping in the cortex**: nearby cells in cortex represented nearby regions in the visual field.

# Hierarchical organization



# Neurocognitron

**Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position** by Kunihiko Fukushima, 1980.

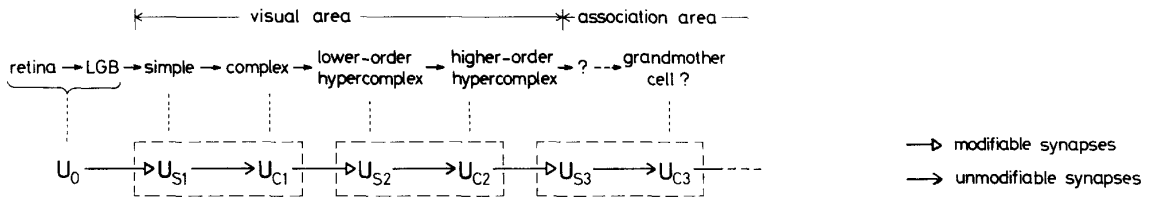
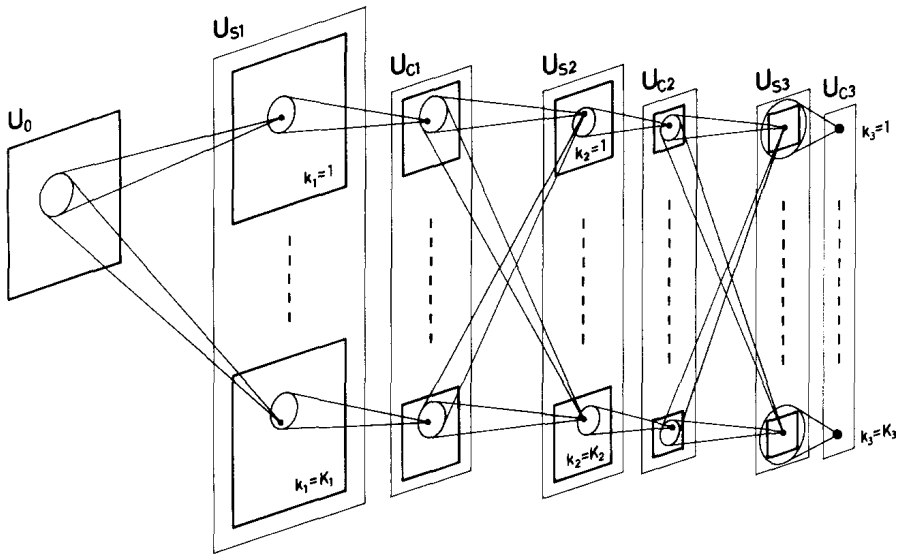
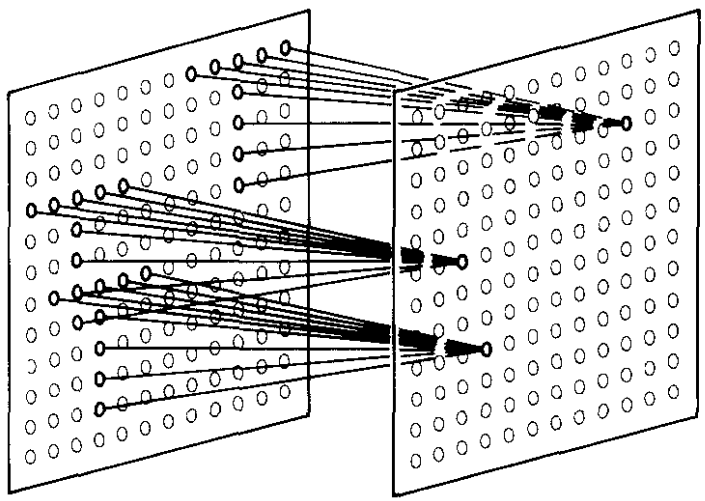



**Fig. 1.** Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

## Inspired by Hubel & Wiesel model

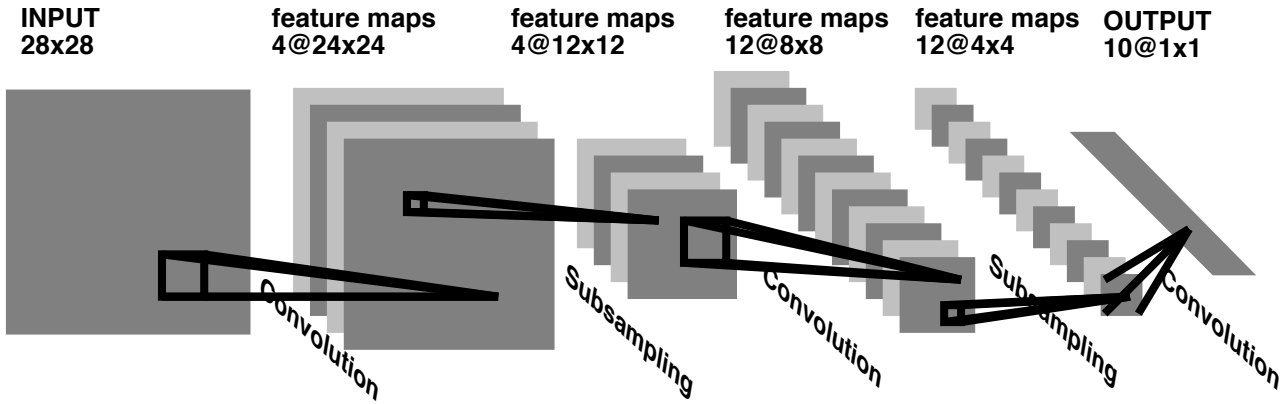


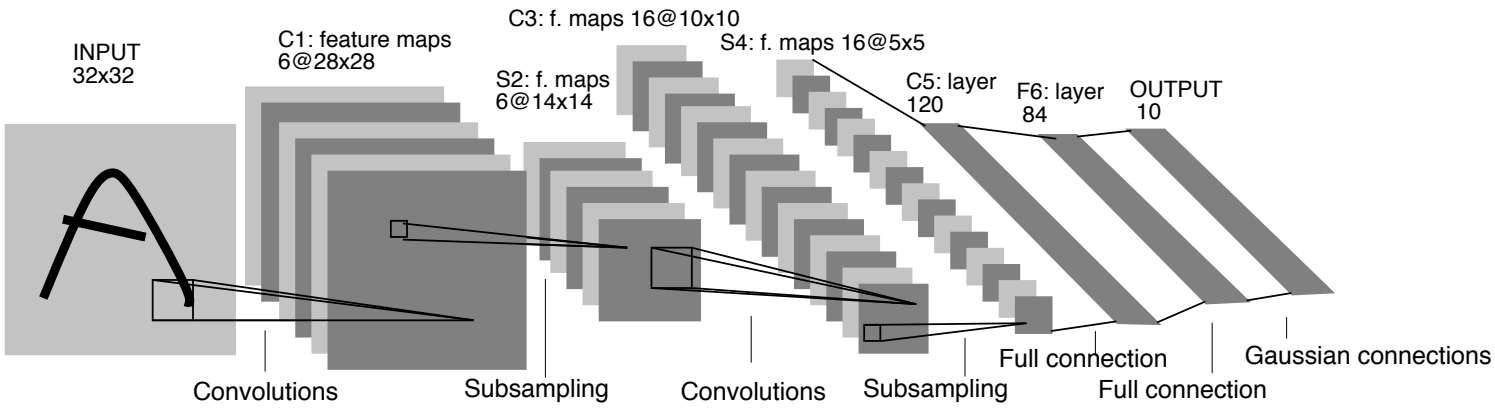
*sandwich* architecture (SCSCSC...)

**simple cells**: modifiable parameters, **complex cells**: perform pooling

# LeCun's *LeNet* ConvNets

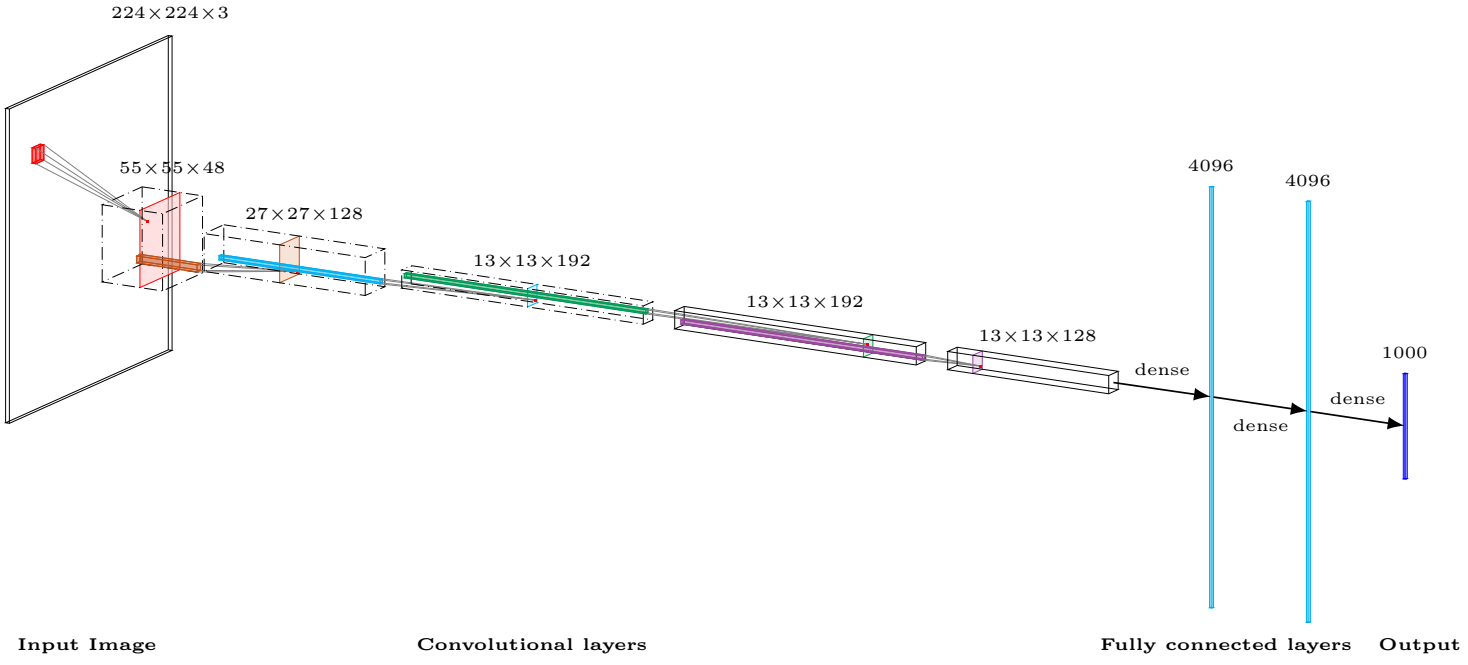



**LeNet 1 '90**




**LeNet 5 '95**

# AlexNet

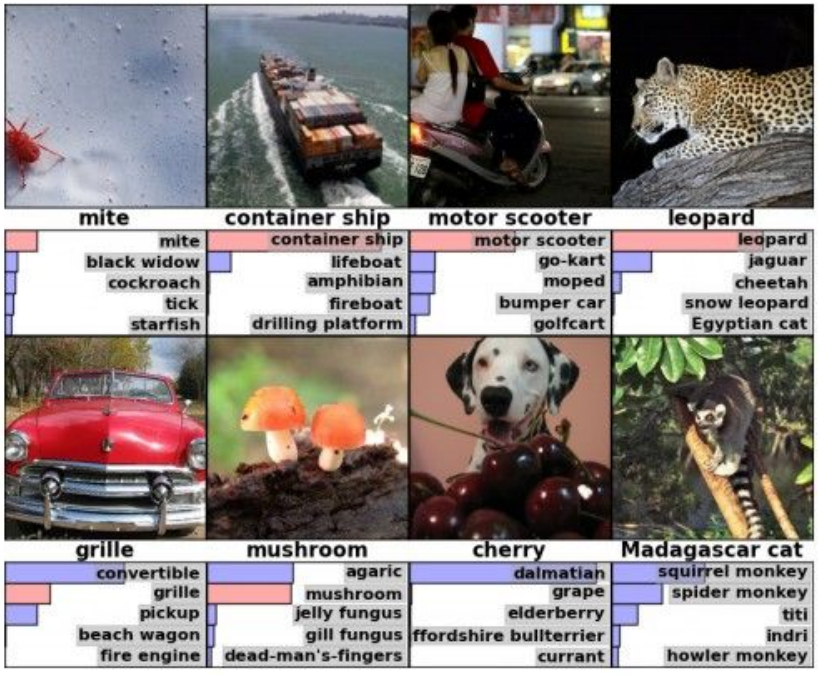



**ImageNet Classification with Deep Convolutional Neural Networks** by Krizhevsky, Sutskever, Hinton, 2012

# Fast-forward to today: ConvNets are everywhere

Classification

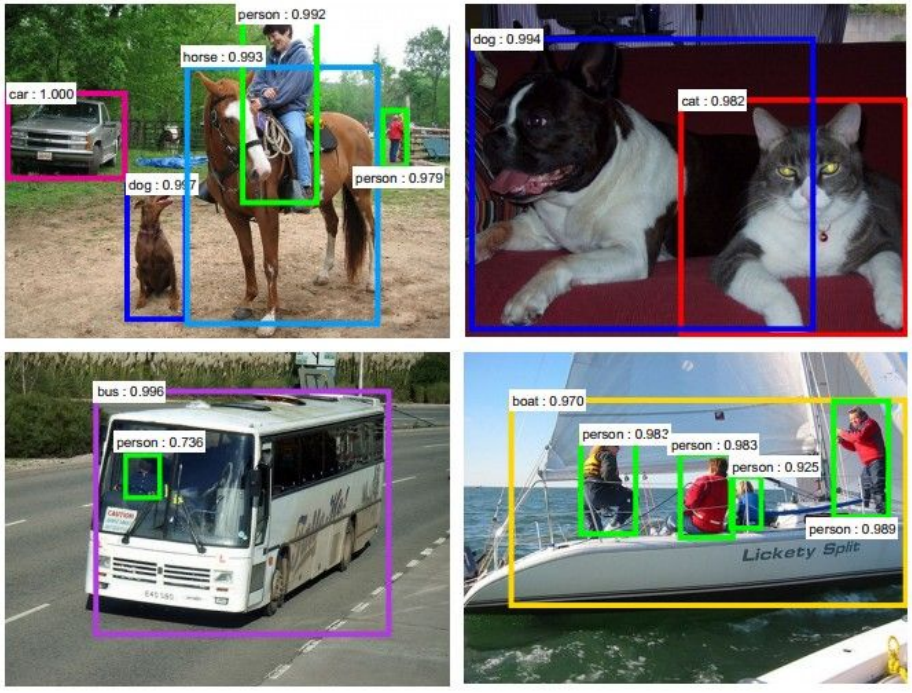
Retrieval




*[Krizhevsky 2012]*

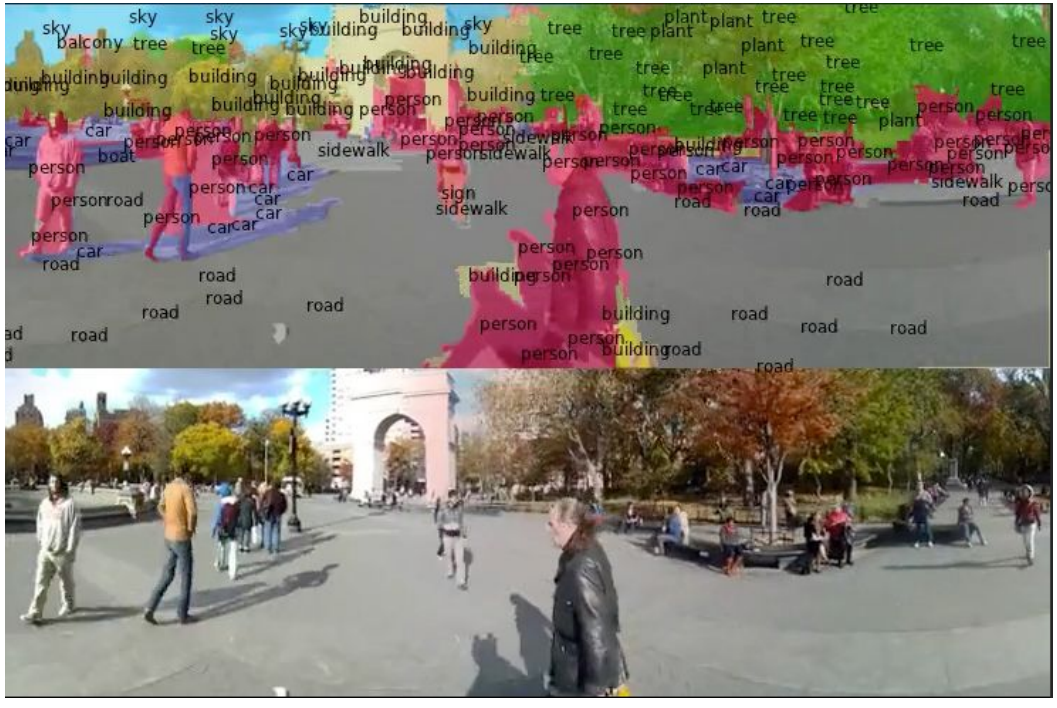
# Fast-forward to today: ConvNets are everywhere

Detection




*[Faster R-CNN: Ren, He, Girshick, Sun 2015]*

Segmentation




*[Farabet et al., 2012]*

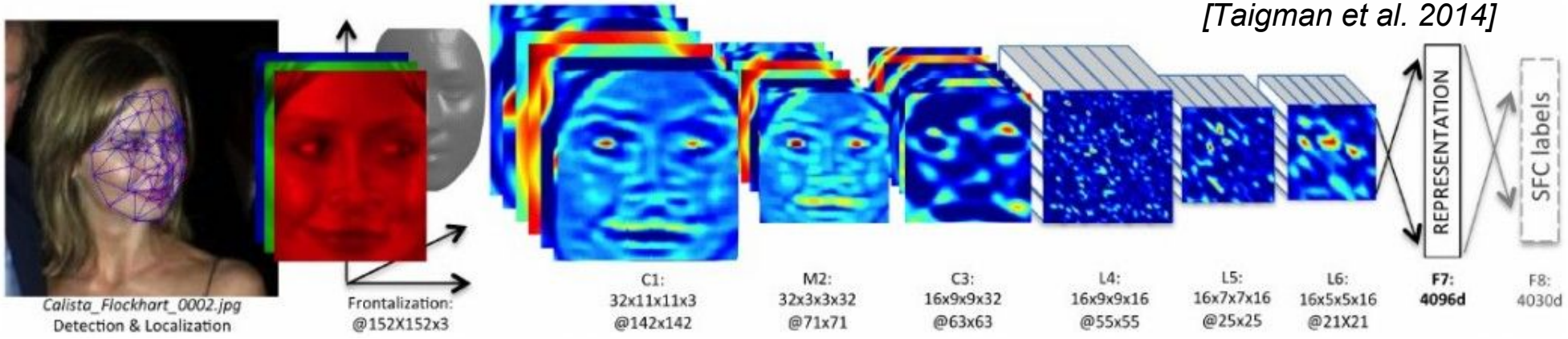
# Fast-forward to today: ConvNets are everywhere

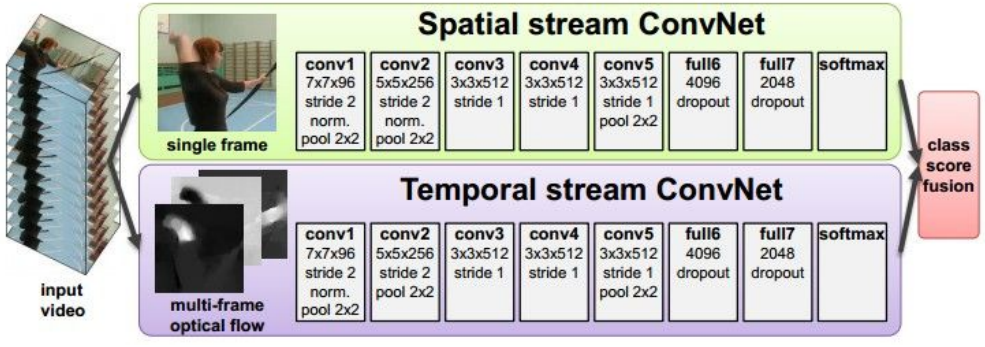

self-driving cars


NVIDIA Tegra X1

# Fast-forward to today: ConvNets are everywhere

*[Taigman et al. 2014]*




*[Simonyan et al. 2014]*

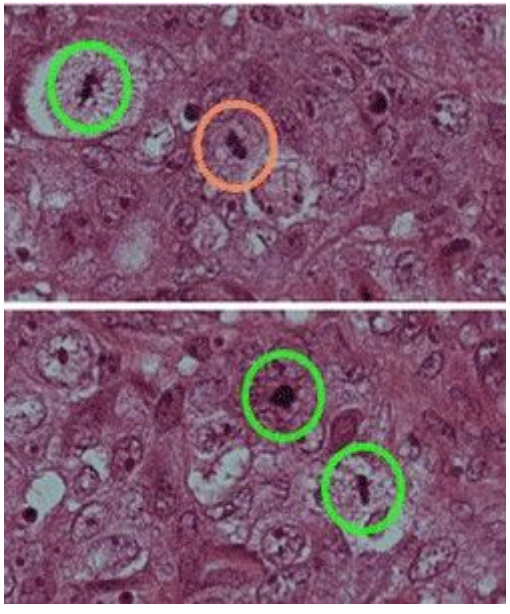
*[Goodfellow 2014]*

# Fast-forward to today: ConvNets are everywhere

*[Toshev, Szegedy 2014]*

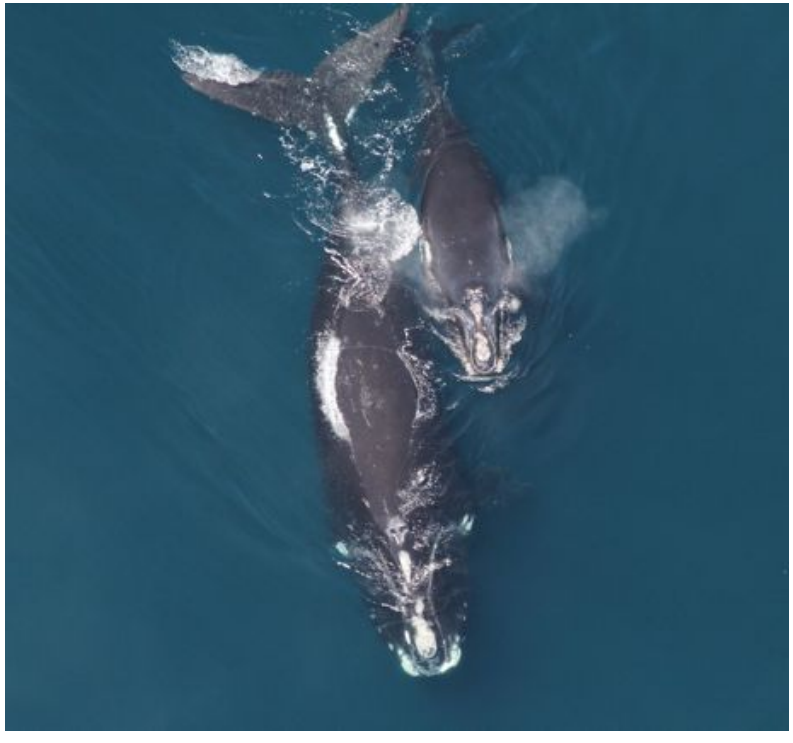
*[Mnih 2013]*

# Fast-forward to today: ConvNets are everywhere



*[Ciresan et al. 2013]*

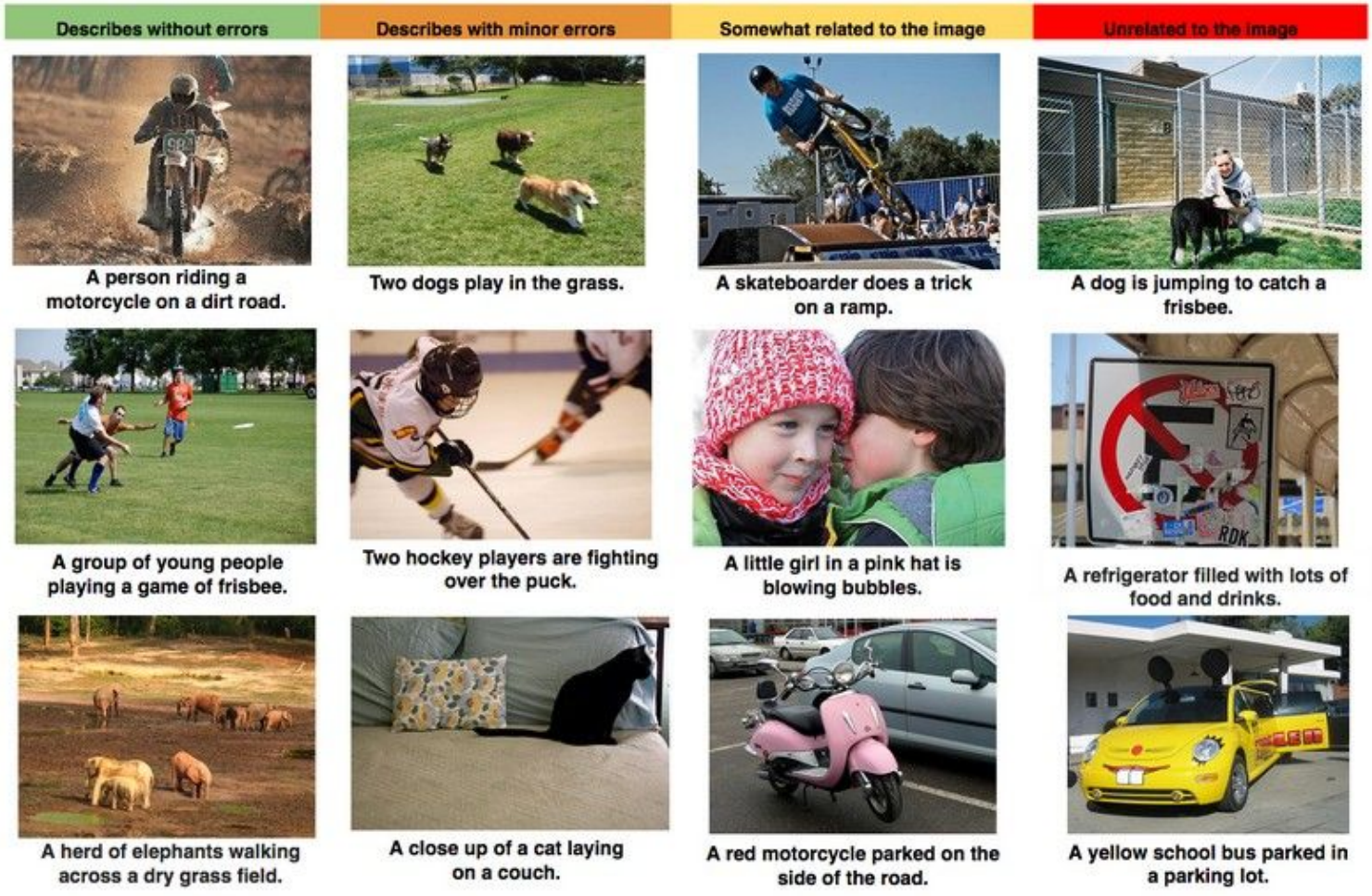
*[Sermanet et al. 2011]*
*[Ciresan et al.]*



*Whale recognition, Kaggle Challenge*

*Mnih and Hinton, 2010*

# Image Captioning




*[Vinyals et al., 2015]*