

“Deep Learning Lecture”

# Lecture 6 : Regularization and Optimization

Liang Wang

Center for Research on Intelligent Perception and Computing (CRIPAC)

National Laboratory of Pattern Recognition (NLPR)

Institute of Automation, Chinese Academy of Science (CASIA)

# Outline

---

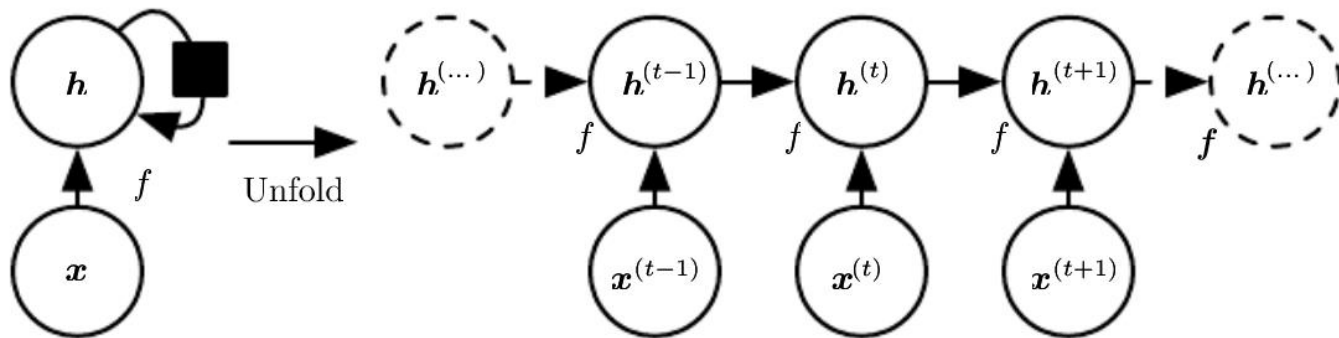
- 1/** Course Review
- 2/** Regularization
- 3/** Optimization
- 4/** Project Grouping

# Review: What is an RNN

- We can consider the states to be the hidden units of the network, so we replace  $s^{(t)}$  by  $h^{(t)}$

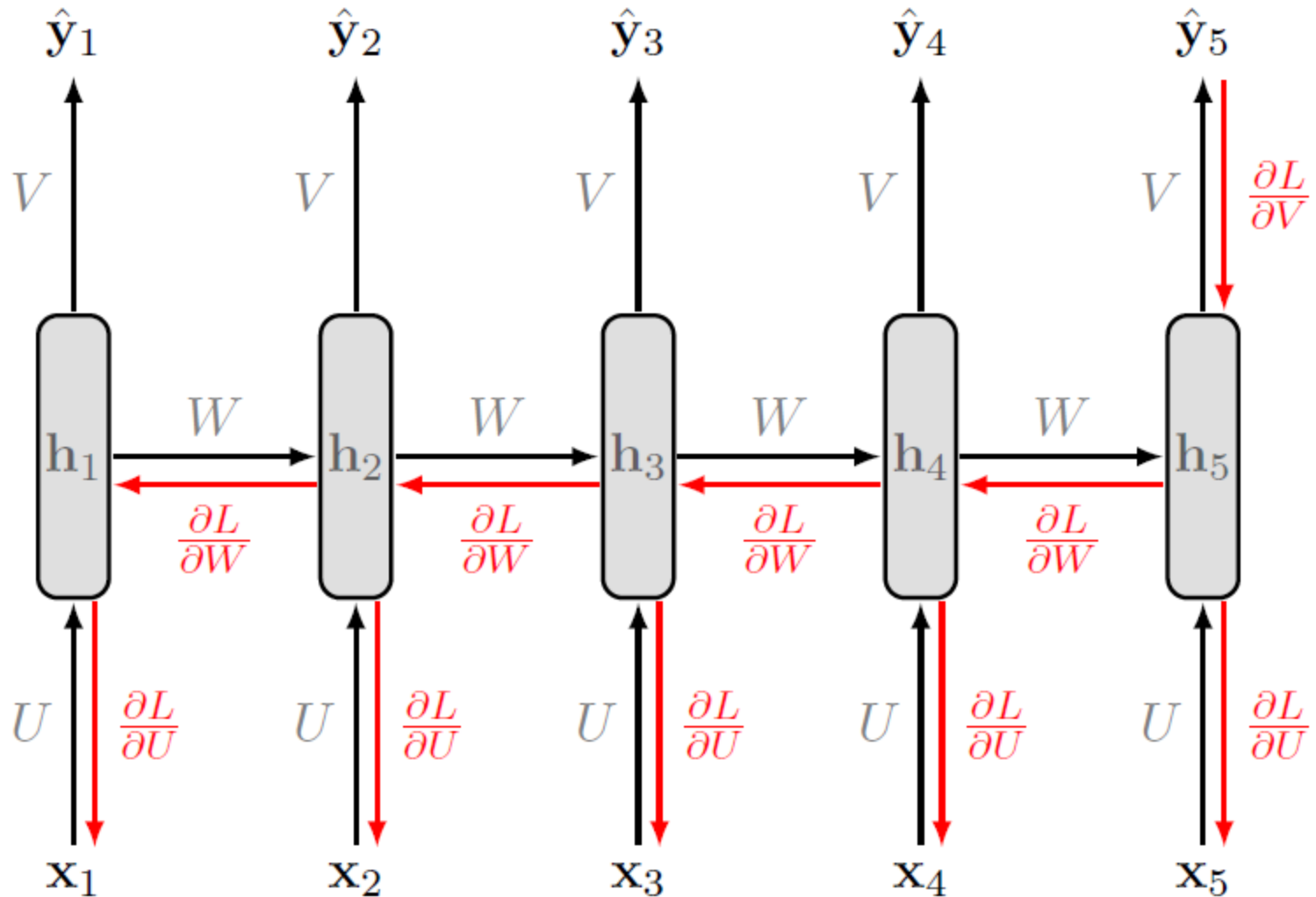
$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

- This system can be drawn in two ways:



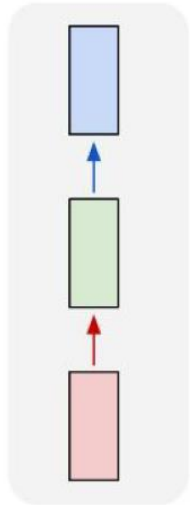
- We can have additional architectural features: Such as output layers that read information from  $h$  to make predictions

# Review: BPTT

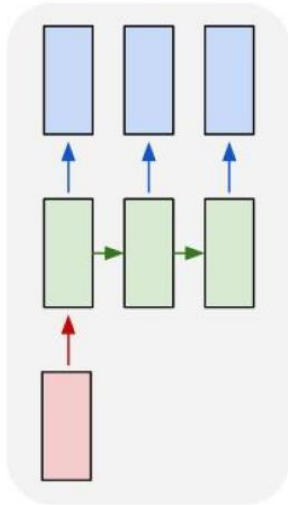


# Review: RNN Paradigms

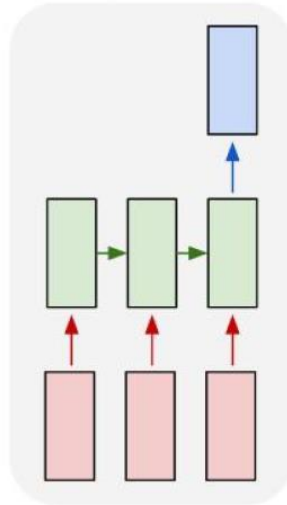
one to one



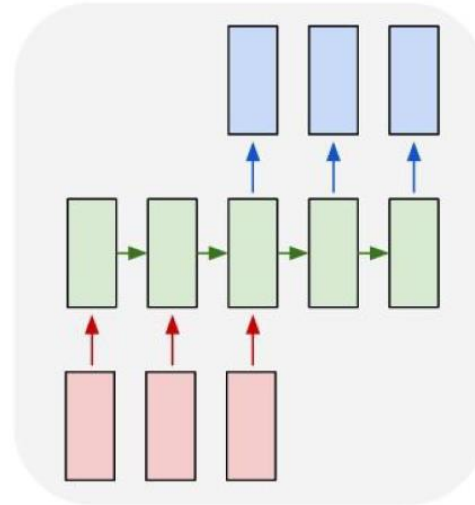
one to many



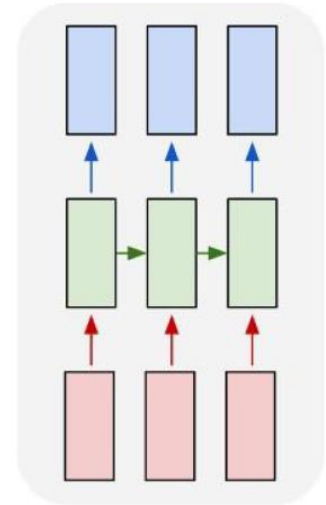
many to one



many to many



many to many



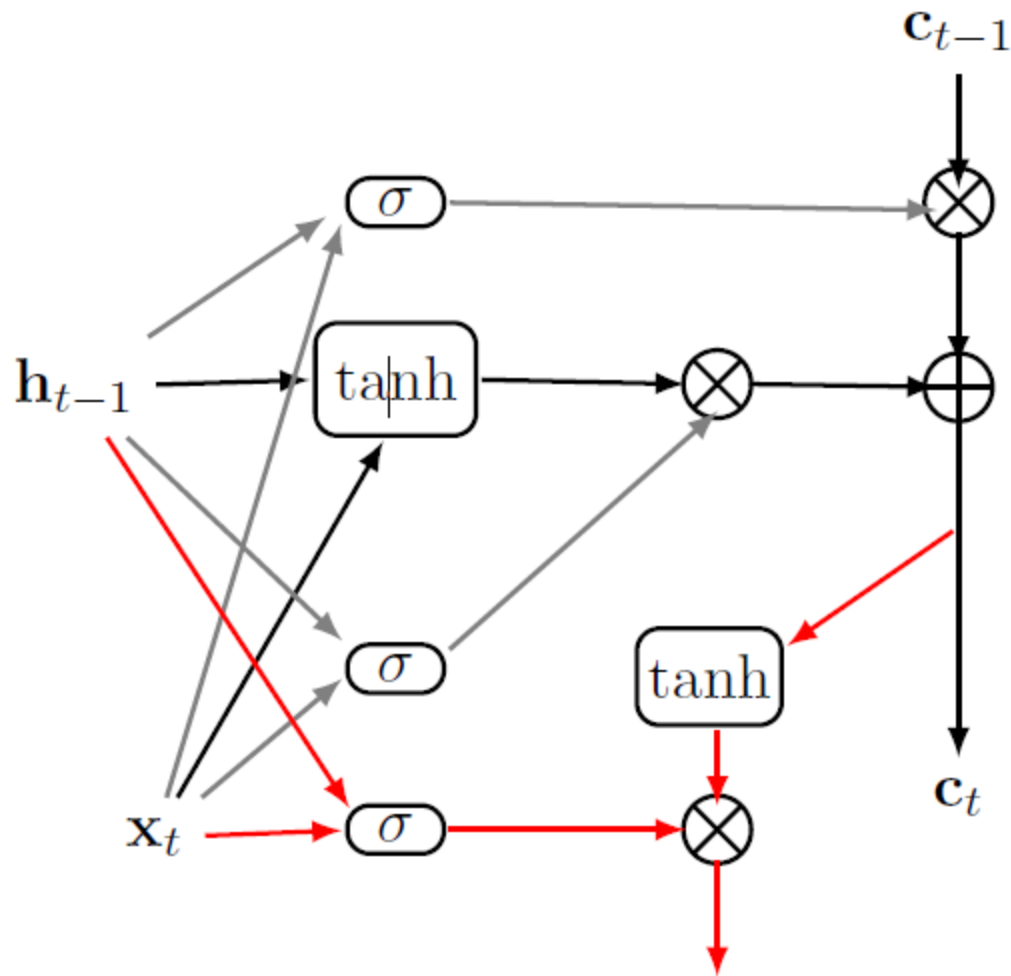
Different problems are more suited for different RNN paradigms

# Review: Challenge of Long-Term Dependencies



- Basic problem: Gradients propagated over many stages tend to vanish (most of the time) or explode (relatively rarely)
- Difficulty with long term interactions (involving multiplication of many jacobians) arises due to exponentially smaller weights, compared to short term interactions
- The problem was first analyzed by Hochreiter and Schmidhuber 1991 and Bengio et al 1993 Lecture

# Review: Long Short Term Memory



$$f_t = \sigma(W_f \mathbf{h}_{t-1} + U_f \mathbf{x}_t)$$

$$i_t = \sigma(W_i \mathbf{h}_{t-1} + U_i \mathbf{x}_t)$$

$$o_t = \sigma(W_o \mathbf{h}_{t-1} + U_o \mathbf{x}_t)$$

$$\tilde{\mathbf{c}}_t = \tanh(W \mathbf{h}_{t-1} + U \mathbf{x}_t)$$

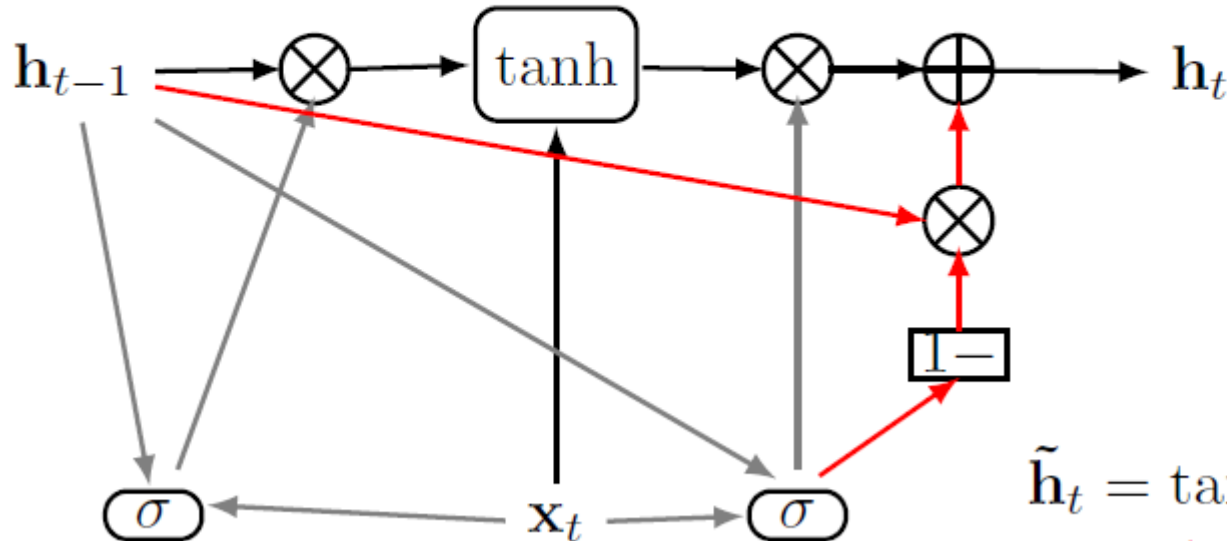
$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + i_t \odot \tilde{\mathbf{c}}_t$$

$$\mathbf{h}_t = o_t \odot \tanh(\mathbf{c}_t)$$

# Review: Gated Recurrent Unit

$$r_t = \sigma(W_r \mathbf{h}_{t-1} + U_r \mathbf{x}_t)$$

$$z_t = \sigma(W_z \mathbf{h}_{t-1} + U_z \mathbf{x}_t)$$

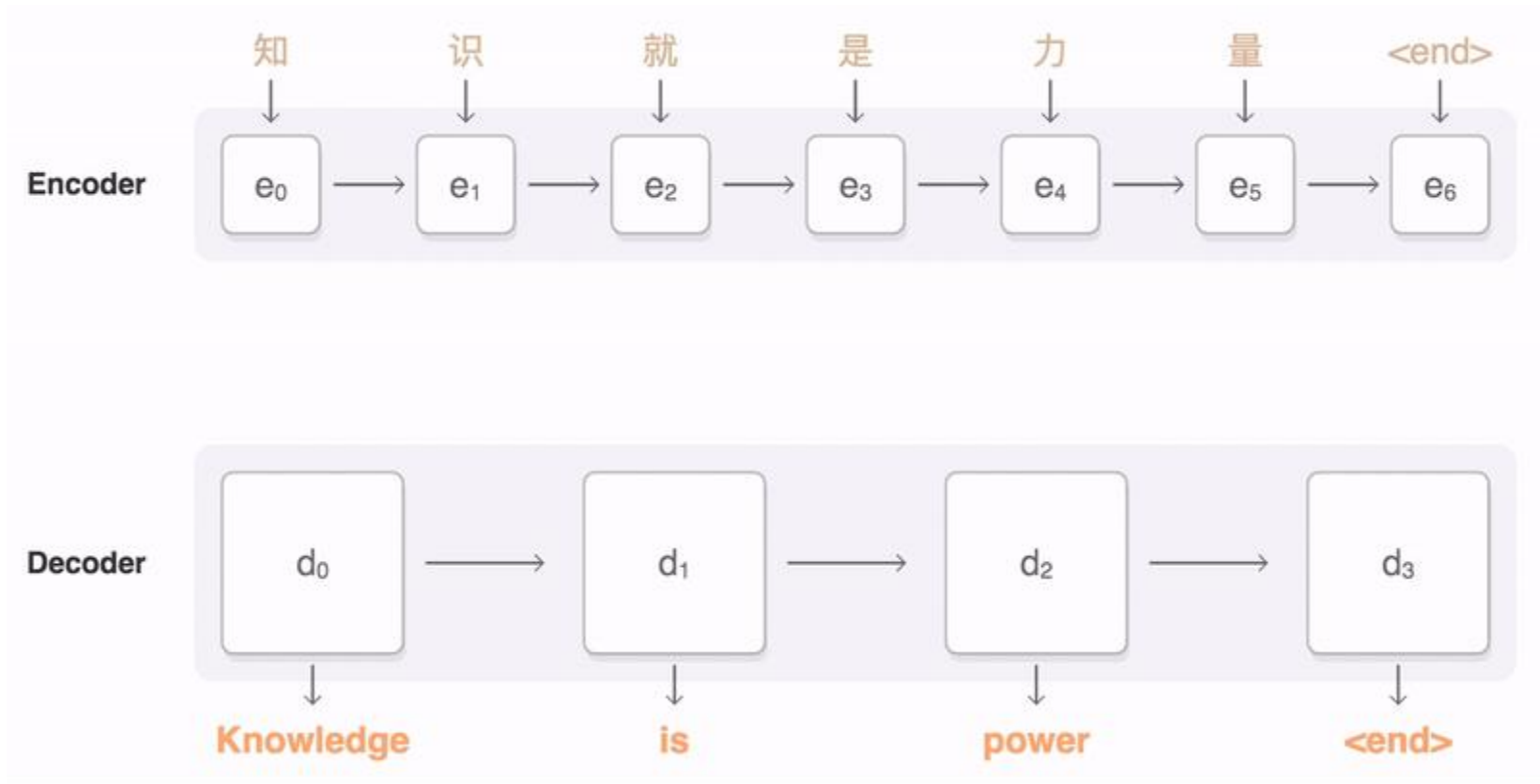


$$\tilde{\mathbf{h}}_t = \tanh(W(r_t \odot \mathbf{h}_{t-1}) + U \mathbf{x}_t)$$

$$\mathbf{h}_t = (1 - z_t) \odot \mathbf{h}_{t-1} + z_t \odot \tilde{\mathbf{h}}_t$$



# Review: Neural Machine Translation



A general-purpose encoder-decoder framework for  
Machine Translation

# Review: Image Captioning Using Attention

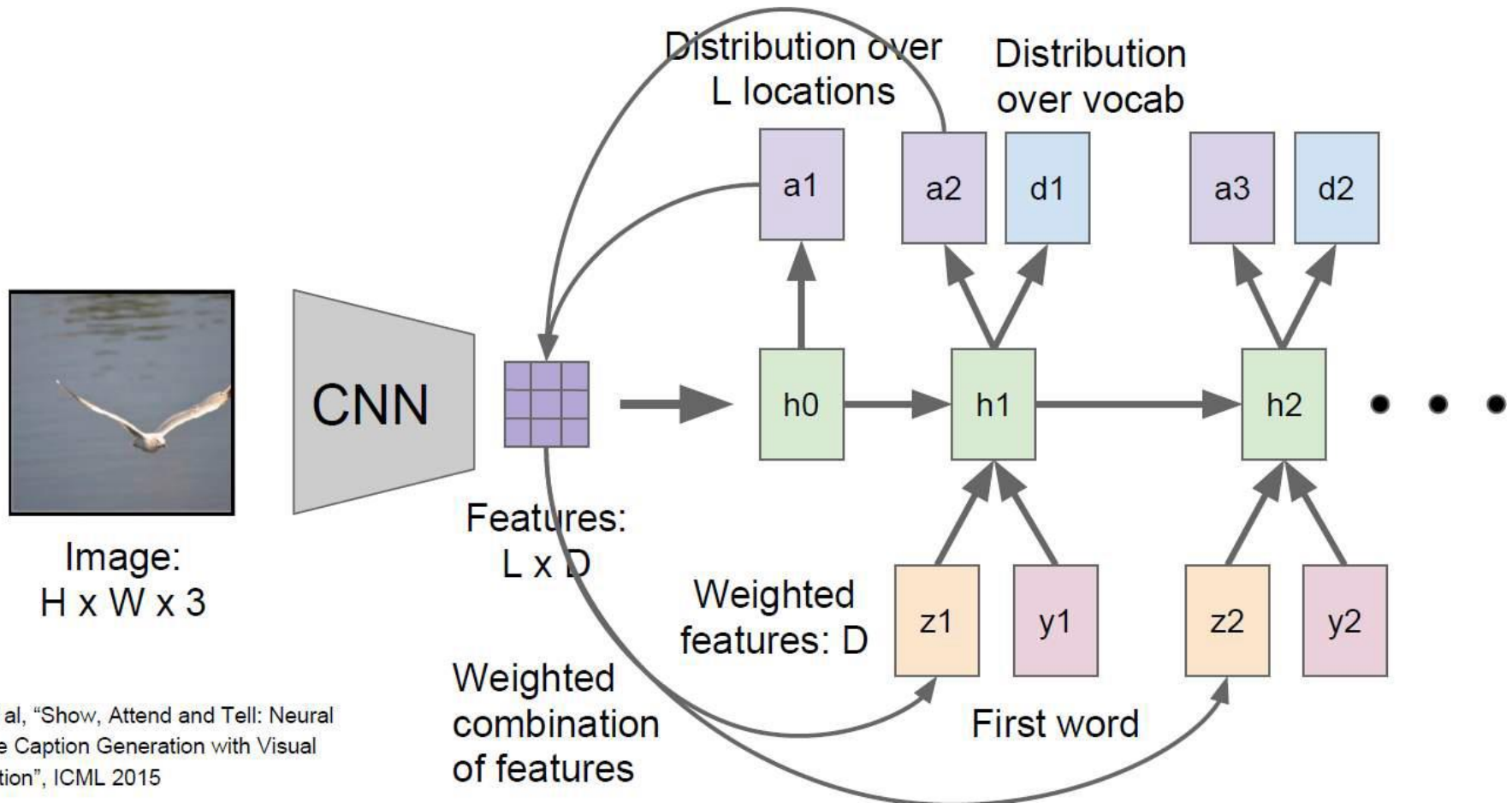


Figure: Andrej Karpathy

# Outline

---

**1/** Course Review

**2/** Regularization

**3/** Optimization

**4/** Project Grouping

# Introduction



- **Regularization** is any modification made to the learning algorithm with an intention to lower the generalization error but not the training error.
- Many standard regularization concepts from machine learning can be readily extended to deep models
- In context of deep learning, most regularization strategies are based on **regularizing estimators**. This is done through reducing variance at the expense of increasing the bias of the estimator.
- An effective regularizer is one that decreases the variance significantly while not overly increasing the bias.

# Introduction



- We discussed three regimes concerning the capacity of models where the model either:
  - Excludes the true data generating process which induces bias (underfitting).
  - Matches the true data generating process.
  - Includes the true data generating process, but also includes many other possible candidates, which results in variance dominating the estimation error (overfitting).
- The goal of regularization is to take the model from the **third** to the **second** regime.

# Motivation

---

- In practice, we never have access to the true data generating distribution. This is a direct result of the extremely complicated domains (images, text and audio sequences) we work with when applying deep learning algorithms.
- In most applications of deep learning, the data generating process is almost certainly **outside the chosen model family**.
- All of the above implies that controlling the complexity of the model is not a simple matter of finding the right model size and the right number of parameters.
- Instead, deep learning relies on finding the best fitting model as a large model that has been **regularized** properly.

# **Classical Regularization Strategies**

# Parameter Norm Penalties

---

- The most traditional form of regularization applicable to deep learning is the concept of **parameter norm penalties**.
- This approach limits the capacity of the model by adding the penalty  $\Omega(\theta)$  to the objective function resulting in:

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(\theta)$$

- $\alpha \in [0,1)$  is a hyperparameter that weights the relative
- contribution of the norm penalty to the value of the objective
- function.



# Parameter Norm Penalties

---

- When the optimization procedure tries to minimize the objective function, it will also decrease some measure of **size** of the parameters .
- Note: The bias terms in the affine transformations of deep models usually require less data to be fit and are usually left unregularized.
- Without loss of generality, we will assume we will be regularizing only the weights **w**.

# L2 Norm Parameter Regularization

---

- The L2 parameter norm penalty, also known as **weight decay** drives **w** closer to the origin by adding the regularization term:

$$\Omega(\mathbf{w}) = \frac{1}{2} ||\mathbf{w}||_2^2 = \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

- For now, assume there is no bias parameters, only weights.

# L2 Norm Parameter Regularization

---

- The update rule of gradient decent using L2 norm penalty is:

$$\mathbf{w} \leftarrow (1 - \epsilon\alpha)\mathbf{w} - \epsilon\nabla_w J(\mathbf{w})$$

- The weights **multiplicatively shrink** by a constant factor at each step.

# L1 Norm Parameter Regularization

---

- L1 norm is another option that can be used to penalize the size of model parameters.
- L1 regularization on the model parameters  $\mathbf{w}$  is:

$$\Omega(\theta) = ||\mathbf{w}|| = \sum_i |w_i|$$

- What is the difference between L2 and L1 norm penalty when applied to machine learning models ? But what happens over the entire course of training in both?

# L1 Norm Parameter Regularization

---

- On the other hand, the L1 norm penalty provides solutions that are **sparse**.
- This sparsity property can be thought of as a feature selection mechanism.

# Conclusion

---

- **L2 norm** penalty can be interpreted as a MAP Bayesian Inference with a **Gaussian** prior on the weights.
- On the other hand, **L1 norm** penalty can be interpreted as a MAP Bayesian Inference with a **Isotropic Laplace Distribution** prior on the weights.

# Dataset Augmentation



- We have seen that for consistent estimators, the best way to get better generalization is to train on more data.
- The problem is that under most circumstances, data is limited. Furthermore, labelling is an extremely tedious task.
- **Dataset Augmentation** provides a cheap and easy way to increase the amount of your training data.
- Certain tasks such as steering angle regression require dataset augmentation to perform well.

# Dataset Augmentation: Color jitter

---





# Dataset Augmentation: Color jitter

- **Color jitter** is a very effective method to augment datasets. It is also extremely easy to apply.
- **Fancy PCA** was proposed by Krizhevsky *et al.* in the famous Alex net paper. It is a way to perform color jitter on images.
- Fancy PCA Algorithm:
  - Perform PCA on the three color channels of your entire dataset.
  - From the covariance matrix provided by PCA, extract the eigenvalues  $\lambda_1, \lambda_2, \lambda_3$  and their corresponding eigenvectors  $p_1, p_2, p_3$ .
  - Add  $pi [a_1 \lambda_1, a_2 \lambda_2, a_3 \lambda_3]^T$  to the  $i^{\text{th}}$  color channel.  $a_1 \dots a_3$  are random variables sampled for each augmented image from a zero mean Gaussian distribution with a variance of 0.1.

# Dataset Augmentation: Horizontal Flipping



# Dataset Augmentation: Horizontal Flipping

---

- **Horizontal Flipping** is applied on data that exhibit horizontal asymmetry.
- Care must be taken to propagate the labels through this transformation.
- Horizontal flipping can be applied to natural images and point clouds. Essentially, one can double the amount of data through horizontal flipping.

# Dataset Augmentation: Conclusion

---

- Many other task specific dataset augmentation algorithms exist. It is highly advised to always use dataset augmentation.
- However, be careful not to alter the correct output!
- Example: b and d, horizontal flipping.
- Furthermore, when comparing two machine learning algorithms train both with either augmented or non-augmented dataset. Otherwise, no subjective decision can be made on which algorithm performed better.

# Noise Robustness



- **Noise Injection** can be thought of as a form of regularization. The addition of noise with infinitesimal variance at the input of the model is equivalent to imposing a penalty on the norm of the weights (Bishop, 1995).
- Noise can be injected at different levels of deep models.

# Noise Robustness: Noise Injection on Weights

---

- Noise added to weights can be interpreted as a more traditional form of regularization.
- This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output.
- In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions (Hochreiter and Schmidhuber, 1995).

# Noise Robustness: Noise Injection on Outputs

---

- Most datasets have some amount (A LOT!) of mistakes in the  $y$  labels. Minimizing our cost function on wrong labels can be extremely harmful.
- One way to remedy this is to explicitly model the noise on labels. This is done through setting a probability  $\varepsilon$  *for which* we think the labels are correct.
- This probability is easily incorporated into the cross entropy cost function **analytically**.
- An example is **label smoothing**.

# Noise Robustness : Label Smoothing

---

- Usually, we have output vectors provided to us as
  - $y_{label} = [1, 0, 0, 0 \dots 0]$ .
- Softmax output is usually of the form
  - $y_{out} = [0.87, 0.001, 0.04, 0.1, \dots 0.03]$ .
- Maximum likelihood learning with a softmax classifier and hard targets may actually never converge, the softmax can never predict a probability of exactly 0 or exactly 1, so it will continue to learn larger and larger weights, making more extreme predictions. forever



# Noise Robustness : Label Smoothing

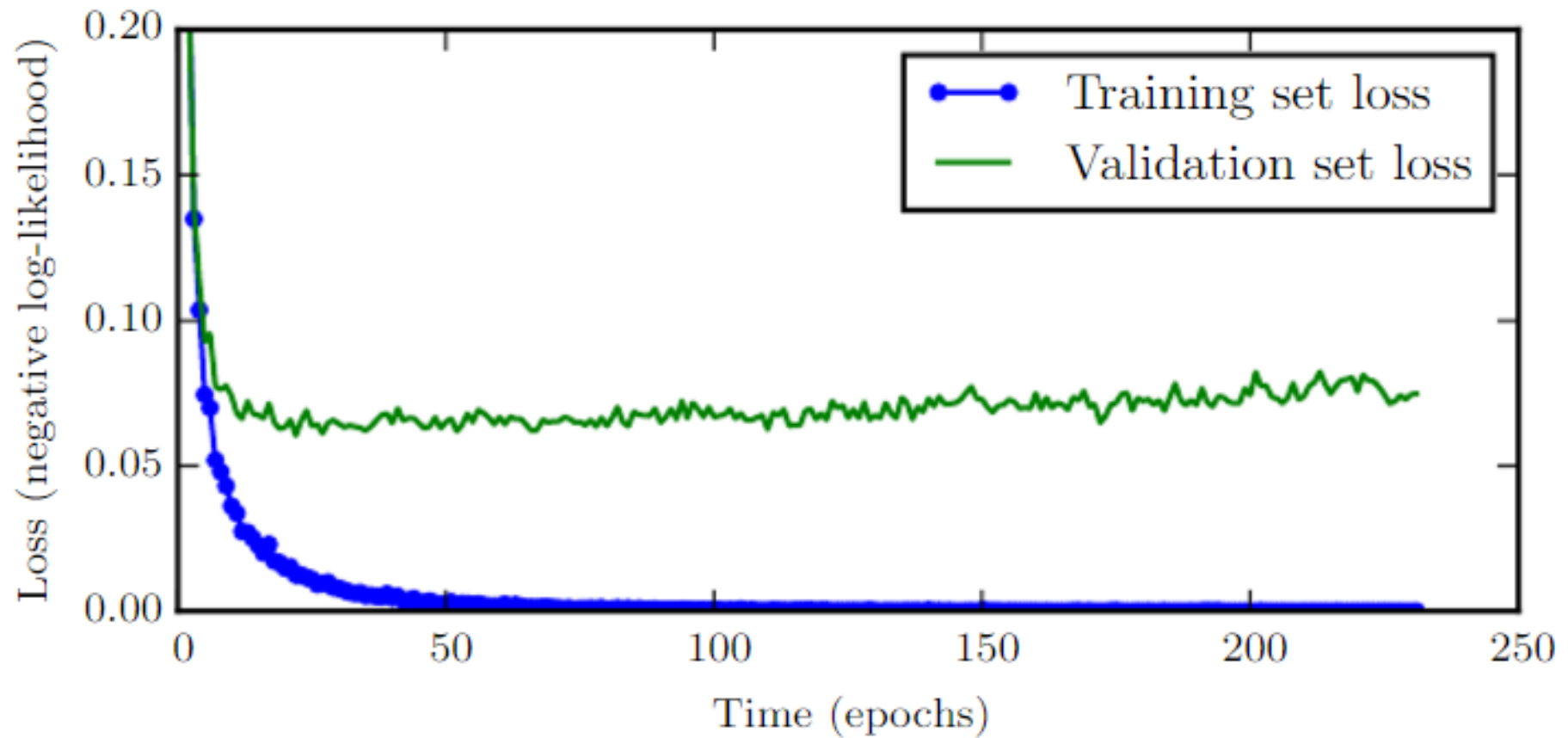
---

- **Label smoothing** replaces the label vector with

$$y_{label} = [1 - \epsilon, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1} \cdots \frac{\epsilon}{K-1}]$$

- The above representation has the advantage of preventing the pursuit of hard probabilities without discouraging correct classification.

# Early Stopping



# Early Stopping: Motivation

---

- When training models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, while the error on the validation set begins to rise again.
- The occurrence of this behaviour in the scope of our applications is almost certain.
- This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error.
- This is termed **Early Stopping**.

# Early Stopping: Meta-Algorithm

---

Let  $n$  be the number of steps between evaluations.

Let  $p$  be the “patience,” the number of times to observe worsening validation set error before giving up.

Let  $\theta_o$  be the initial parameters.

$\theta \leftarrow \theta_o$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

**while**  $j < p$  **do**

    Update  $\theta$  by running the training algorithm for  $n$  steps.

$i \leftarrow i + n$

$v' \leftarrow \text{ValidationSetError}(\theta)$

**if**  $v' < v$  **then**

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

**else**

$j \leftarrow j + 1$

**end if**

**end while**

Best parameters are  $\theta^*$ , best number of training steps is  $i^*$

---

# Early Stopping: Practical Issues

---

- Early Stopping is probably one of the most used regularization strategies in deep learning.
- Early stopping can be thought of as a hyperparameter selection method, where **training time** is the hyperparameter to be chosen.
- However, a portion of data should be reserved for validation.

# Early Stopping: Exploiting the Validation Data

---

- To exploit all of our precious training data we can:
  - Employ early stopping as described above.
  - Retrain using all of the data up to the point that was determined during early stopping.
- Some subtleties arise regarding the definition of **point**.
- Do we train for the same number of parameter updates or for the same number of epochs (passes through training data) ?

# Early Stopping: Exploiting the Validation Data

---

- A second strategy to exploit the full training dataset would be to:
  - Employ early stopping as described above.
  - Continue training with the parameters determined by early stopping, using the validation set data.
- This strategy avoids the high cost of retraining the model from scratch, but is not well-behaved.
- Since we no longer have a validation set, we cannot know if generalization error is improving or not. Our best bet is to stop training when the training error is not decreasing much any more.

# Parameter Sharing



- So far, we have discussed regularization as adding constraints or penalties to the parameters with respect to a **fixed region**.
- However, we might want to express priors on parameters in other ways. Specifically, we might not know which region the parameters would lie in, but rather that there is some dependencies between them.
- Most common type of dependency: Some parameters should be close to each other.



# Parameter Sharing

---

- **Parameter Sharing** imposes much stronger assumptions on parameters through forcing the parameter sets to be **equal**.
- Examples would be Siamese networks, convolution operators, and multitask learning.

# Parameter Tying

---

- **Parameter Tying** refers to explicitly forcing the parameters of two models to be close to each other, through the norm penalty:

$$||\mathbf{w}^{(A)} - \mathbf{w}^{(B)}||$$

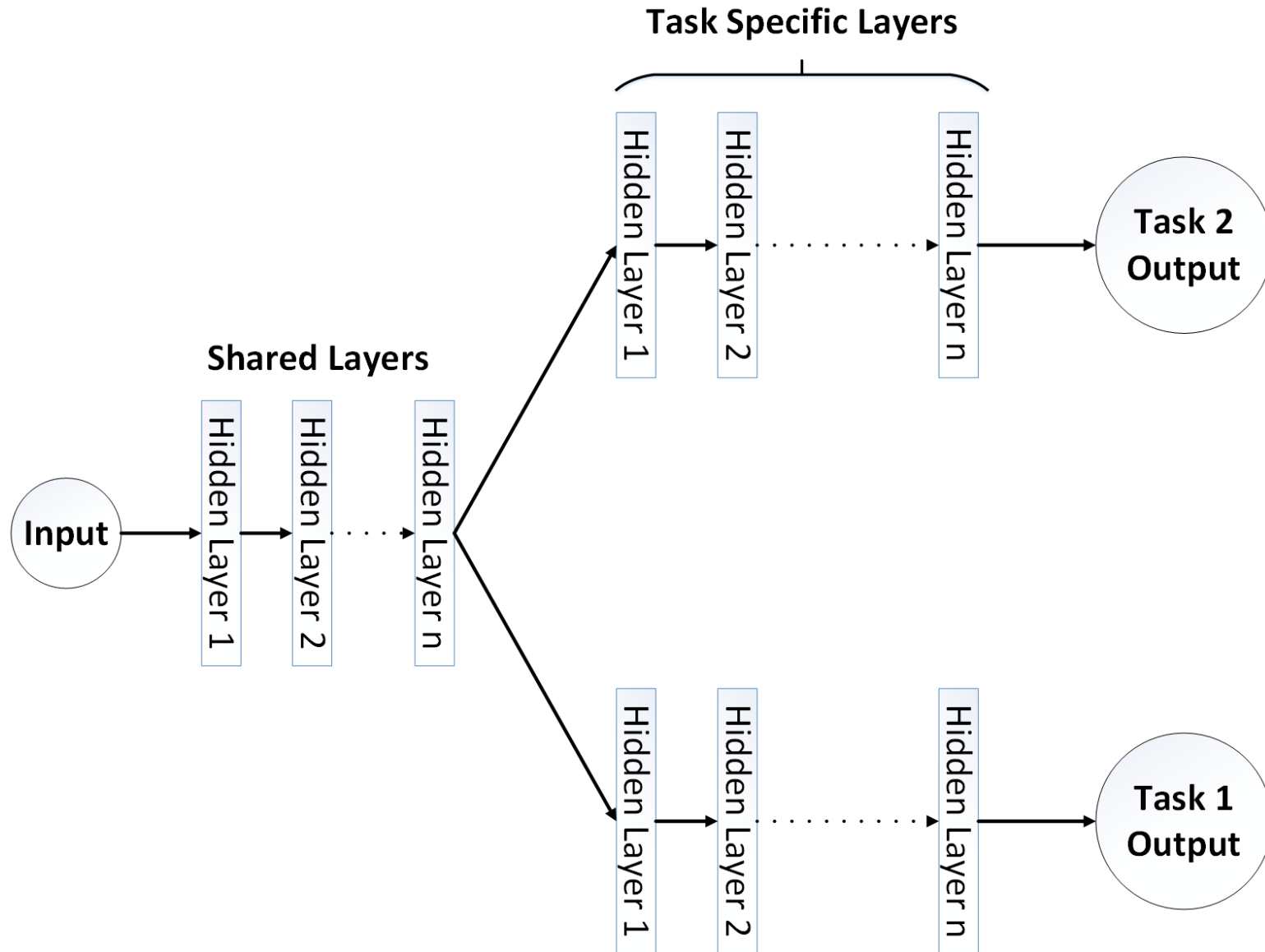
- Here,  $w^{(A)}$  refers to the weights of the first model while  $w^{(B)}$  refers to those of the second one.

# Multitask Learning



- **Multitask Learning** is a way to improve generalization by pooling the examples arising out of several tasks.
- Usually, the most common form of multitask learning is performed through an architecture which is divided to two parts:
  - Task-specific parameters (which only benefit from the examples of their task to achieve good generalization).
  - Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks).
- Multitask learning is a form of parameter sharing.

# Multitask Learning



# Multitask Learning



- Improved generalization and generalization error bounds (Baxter, 1995) can be achieved because of the shared parameters, for which statistical strength can be greatly improved in proportion with the increased number of examples for the shared parameters, compared to the scenario of single-task models.
- Intuitively, the additional task imposes constraints on the parameters in the shared layers, preventing overfitting.
- Improvement in generalization only occurs when there is something shared across the tasks at hand.

# Bagging

---

- **Bagging** (short for **bootstrap aggregating**) is a technique for reducing generalization error through combining several models (Breiman, 1994).
- Bagging is defined as follows:
  - Train  $k$  different models on  $k$  different subsets of training data, constructed to have the same number of examples as the original dataset through random sampling from that dataset with replacement.
  - Have all of the models vote on the output for test examples.
- Techniques employing bagging are called ensemble models.

# Bagging

---

- The reason that Bagging works is that different models will usually not all make the same errors on the test set.
- This is a direct results of training on *k different subsets of the* training data, where each subset is missing some of the examples from the original dataset.
- Other factors such as differences in random initialization, random selection of mini-batches, differences in hyperparameters, or different outcomes of non-deterministic implementations of neural networks are often enough to cause different members of the ensemble to make **partially independent errors**.

# Ensemble Models



- On average, the ensemble will perform at least as well as any of its members, and if the members make independent errors, the ensemble will perform significantly better than its members.
- Proof ?
- The only disadvantage of ensemble models is that they do not provide us with a **scalable** way to improve performance. Usually, ensemble models of more than 2-3 networks become too tedious to train and handle.

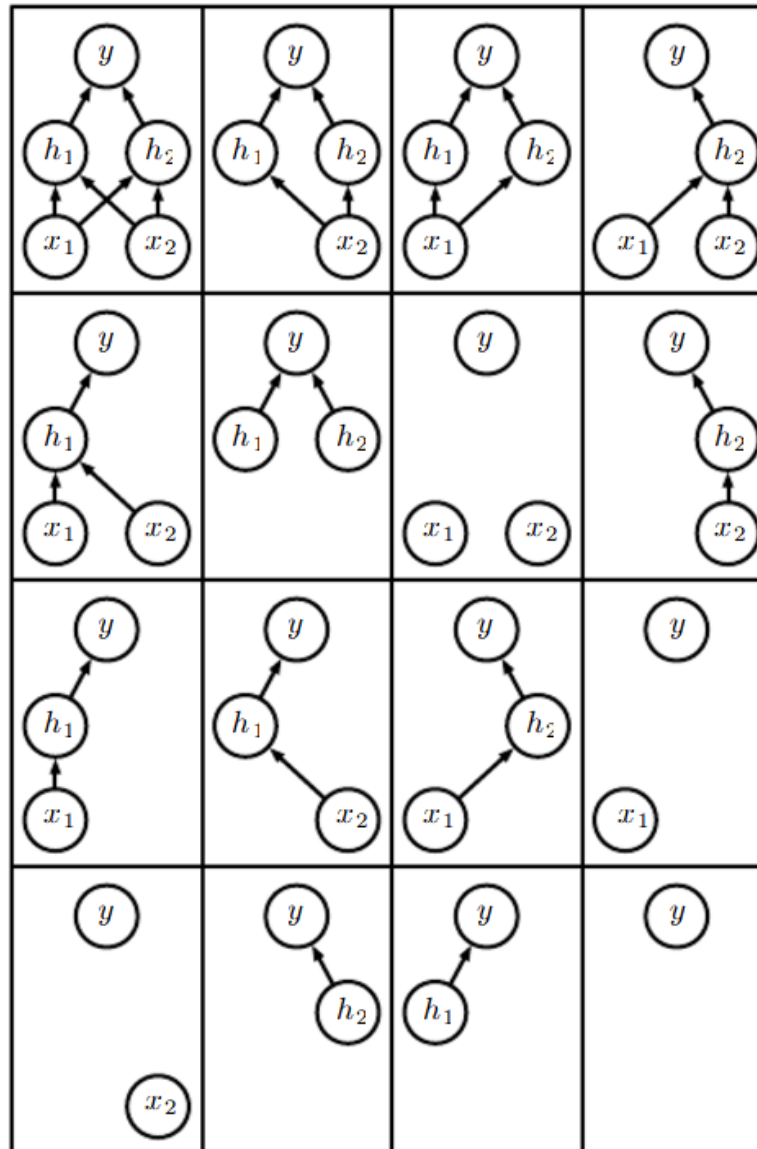
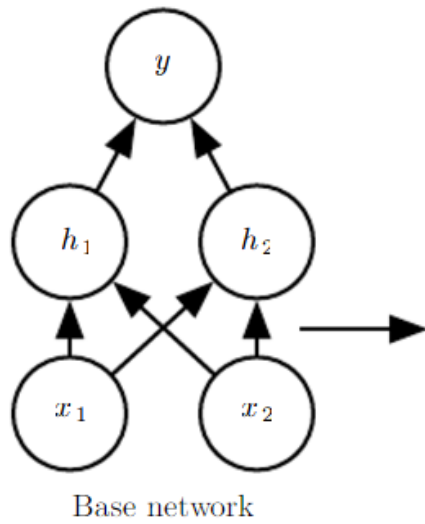


# Dropout



- Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models.
- Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.
- Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network.

# Dropout



Ensemble of subnetworks

# Dropout

---

- Dropout allows us to represent an exponential number of models with a tractable amount of memory.
- Furthermore, Dropout removes the need to accumulate model votes at the inference stage.
- Dropout can intuitively be explained as forcing the model to learn with missing input and hidden units.

# Training with Dropout



- To train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent.
- Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all of the input and hidden units in the network.
- The mask for each unit is sampled independently from all of the others.
- Typically, the probability of including a hidden unit is 0.5, while the probability of including an input unit is 0.8.

# Training with Dropout



- Dropout training has some intricacies we need to be wary of.
- At training time, we are **required** to divide the output of each unit by the probability of that unit's dropout mask.
- The goal is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average.
- No theoretically satisfying basis for the accuracy of this approximate training rule in deep non linear networks, but empirically it performs very well.

# Conclusion

---

- Dropout is that it is very computationally cheap, using dropout during training requires only  $O(n)$  computation per example per update, to generate  $n$  random binary numbers and multiply them by the state.
- Dropout does not significantly limit the type of model or training procedure that can be used. It works well with nearly any model that uses a distributed representation and can be trained with stochastic gradient descent.

# Conclusion

---

- Though the cost per-step of applying dropout to a specific model is negligible, the cost of using dropout in a complete system can be significant.
- Applying Dropout indirectly requires us to design larger systems to preserve capacity. Larger systems usually are slower at inference time.
- Practitioners have to keep in mind that for very large datasets, regularization confers little reduction in generalization error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularization.

# Adversarial Training

---

- In many cases, neural networks have begun to reach human performance when evaluated on an i.i.d. test set.
- However, szegedy *et al.*(2014) found that even networks that have achieved human accuracy, have a 100% error rate on examples that have been intentionally constructed to “fool” the network.
- In many cases, the modified example is so similar to the original one, human observers cannot tell the difference.
- These examples are called **adversarial examples**.



# Adversarial Examples



$x$

$y = \text{"panda"}$   
w/ 57.7%  
confidence

$+ .007 \times$



$\text{sign}(\nabla_x J(\theta, x, y))$

"nematode"  
w/ 8.2%  
confidence

$=$



$x + \epsilon \text{sign}(\nabla_x J(\theta, x, y))$   
"gibbon"  
w/ 99.3 %  
confidence

# Adversarial Examples

---

- Adversarial examples are interesting in the context of regularization because one can reduce the error rate on the original i.i.d. test set via **adversarial training** - training on adversarially perturbed examples from the training set.
- Adversarial training discourages highly sensitive linear behaviour through explicitly introducing a local constancy prior into supervised neural nets.

# Outline

---

**1/** Course Review

**2/** Regularization

**3/** Optimization

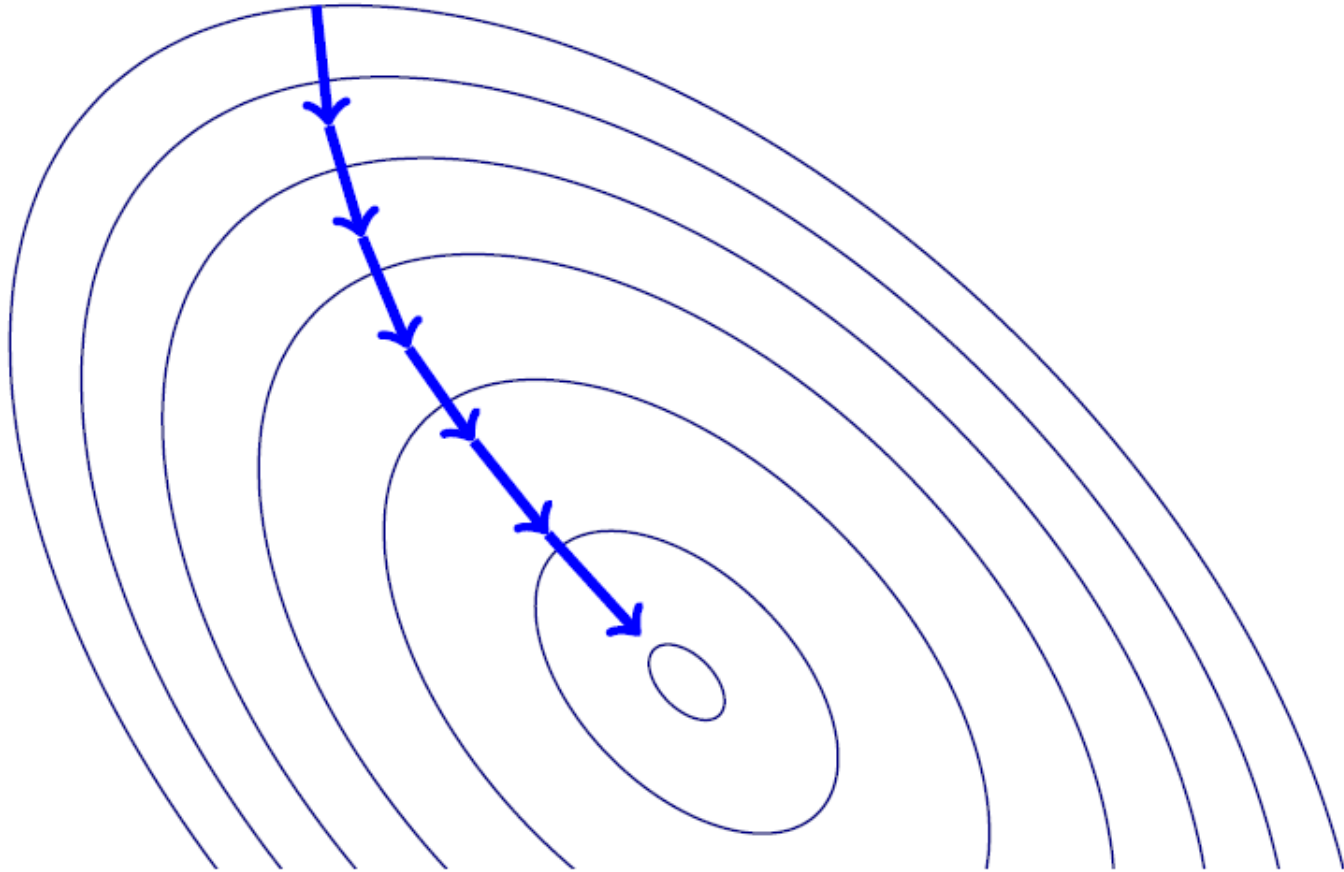
**4/** Project Grouping

# Optimization



- We've seen backpropagation as a method for computing gradients
- Let's see a family of first order methods

# Gradient Descent



# Batch Gradient Descent

---

**Algorithm 1** Batch Gradient Descent at Iteration  $k$

---

**Require:** Learning rate  $\epsilon_k$

**Require:** Initial Parameter  $\theta$

- 1: **while** stopping criteria not met **do**
  - 2: Compute gradient estimate over  $N$  examples:
  - 3:  $\hat{\mathbf{g}} \leftarrow +\frac{1}{N} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
  - 4: Apply Update:
  - 5: **end while**  $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$
- 

- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update

# Stochastic Gradient Descent

---

## Algorithm 2 Stochastic Gradient Descent at Iteration k

---

**Require:** Learning rate  $\epsilon_k$

**Require:** Initial Parameter  $\theta$

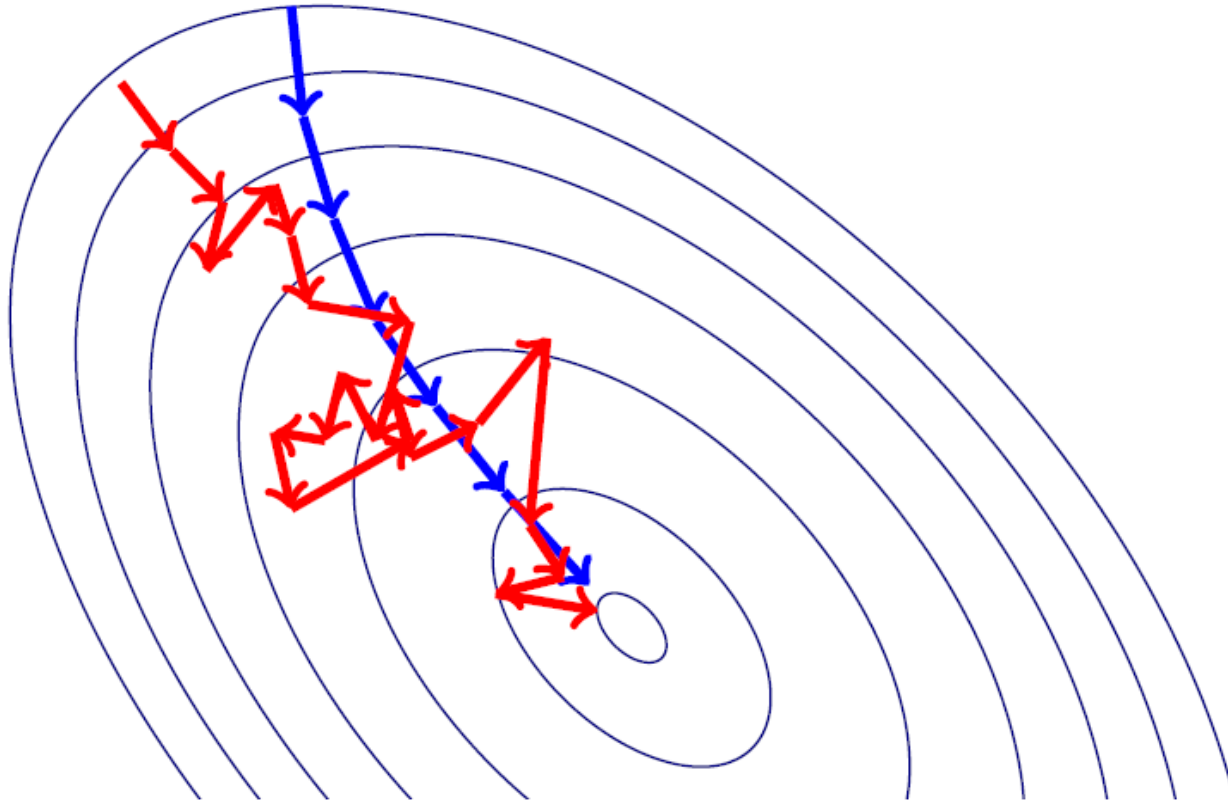
- 1: **while** stopping criteria not met **do**
  - 2: Sample example  $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$  from training set
  - 3: Compute gradient estimate:
  - 4:  $\hat{\mathbf{g}} \leftarrow +\nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
  - 5: **end while**  $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$
- 

- $\epsilon_k$  is learning rate at step k
- Sufficient condition to guarantee convergence:

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

# Stochastic Gradient Descent

---





# Minibatching

---

- **Potential Problem:** Gradient estimates can be very noisy
- **Obvious Solution:** Use larger mini-batches
- **Advantage:** Computation time per update does not depend on number of training examples  $N$
- This allows convergence on extremely large datasets
- See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou

# Learning Rate Schedule

---

- In practice the learning rate is decayed linearly till iteration  $\tau$

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \text{ with } \alpha = \frac{k}{\tau}$$

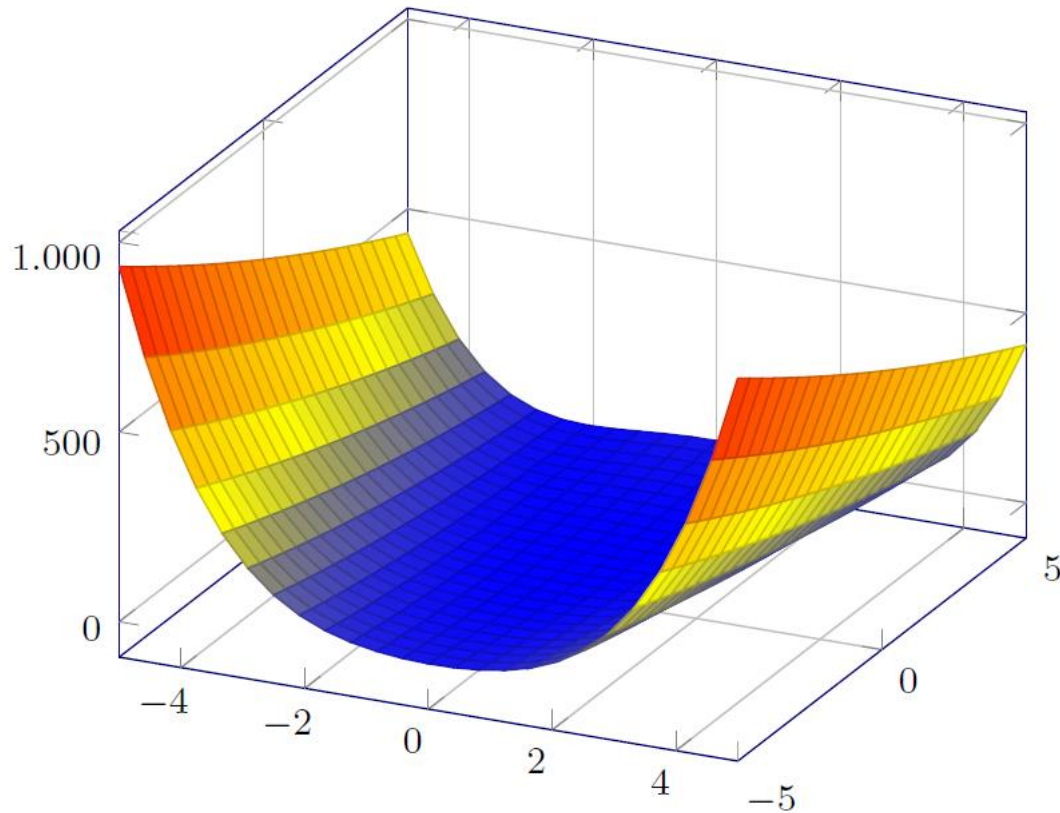
- $\tau$  is usually set to the number of iterations needed for a large number of passes through the data
- $\epsilon_\tau$  should roughly be set to 1% of  $\epsilon_0$
- How to set  $\epsilon_0$  ?

# Momentum



- The Momentum method is a method to accelerate learning using SGD
- In particular SGD suffers in the following scenarios:
  - Error surface has high curvature
  - We get small but consistent gradients
  - The gradients are very noisy

# Momentum



- Gradient Descent would move quickly down the walls, but very slowly through the valley floor

# Momentum



- How do we try and solve this problem?
- Introduce a new variable  $\mathbf{v}$ , the velocity
- We think of  $\mathbf{v}$  as the direction and speed by which the parameters move as the learning dynamics progresses
- The velocity is an **exponentially decaying moving average** of the negative gradients

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

- $\alpha \in [0, 1)$  Update rule:  $\theta \leftarrow \theta + \mathbf{v}$

# Momentum

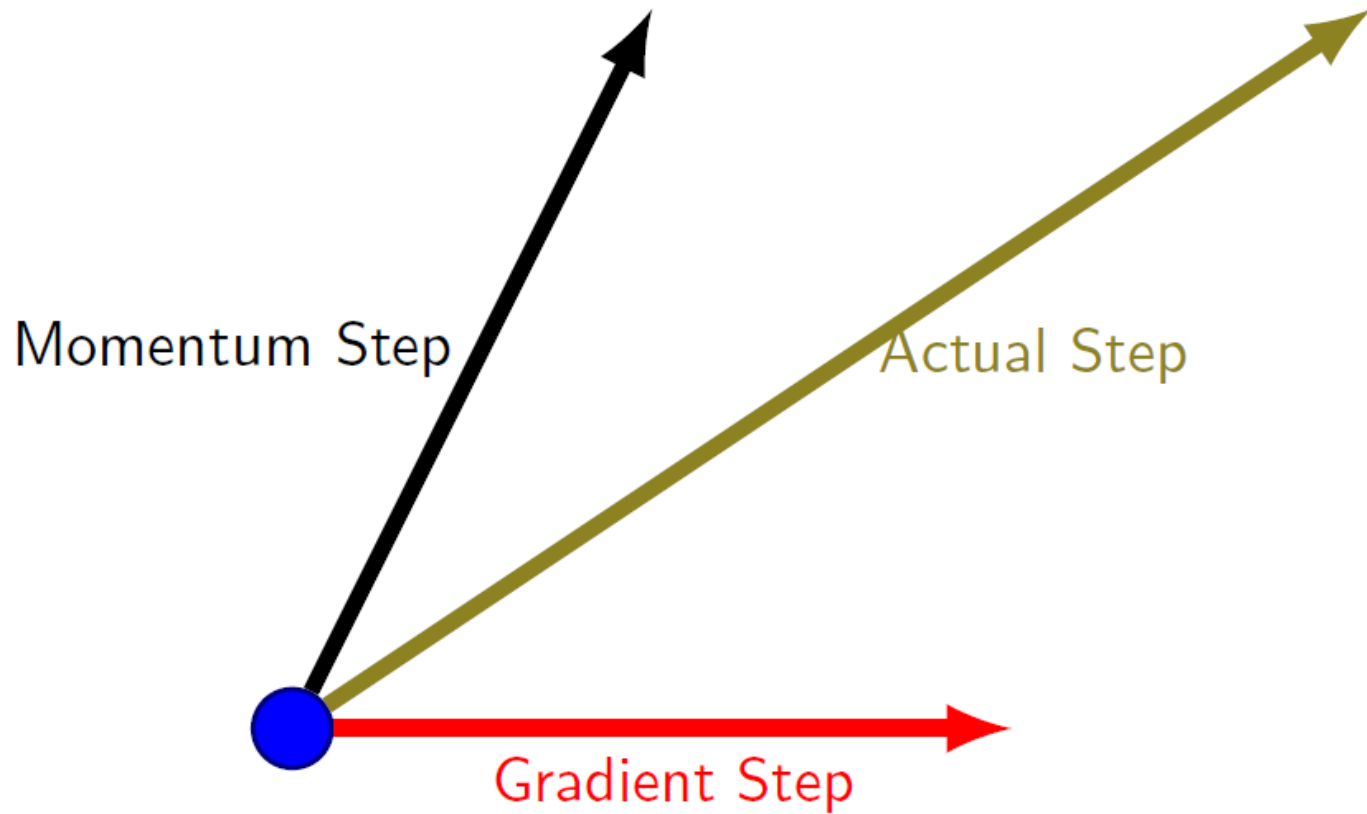
---

- Let's look at the velocity term:

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

- The velocity **accumulates** the previous gradients
- What is the role of  $\alpha$  ?
  - If  $\alpha$  is larger than  $\epsilon$  the current update is more affected by the previous gradients
  - Usually values for  $\alpha$  are set high  $\approx 0.8, 0.9$

# Momentum



# Momentum: Step Sizes

---

- In SGD, the step size was the norm of the gradient scaled by the learning rate:  $\epsilon \|g\|$ .
- While using momentum, the step size will also depend on the norm and alignment of a sequence of gradients
- For example, if at each step we observed  $g$ , the step size would be :

$$\epsilon \frac{\|g\|}{1 - \alpha}$$

- If  $\alpha = 0.9 \Rightarrow$  multiply the maximum speed by 10 relative to the current gradient direction



# Momentum

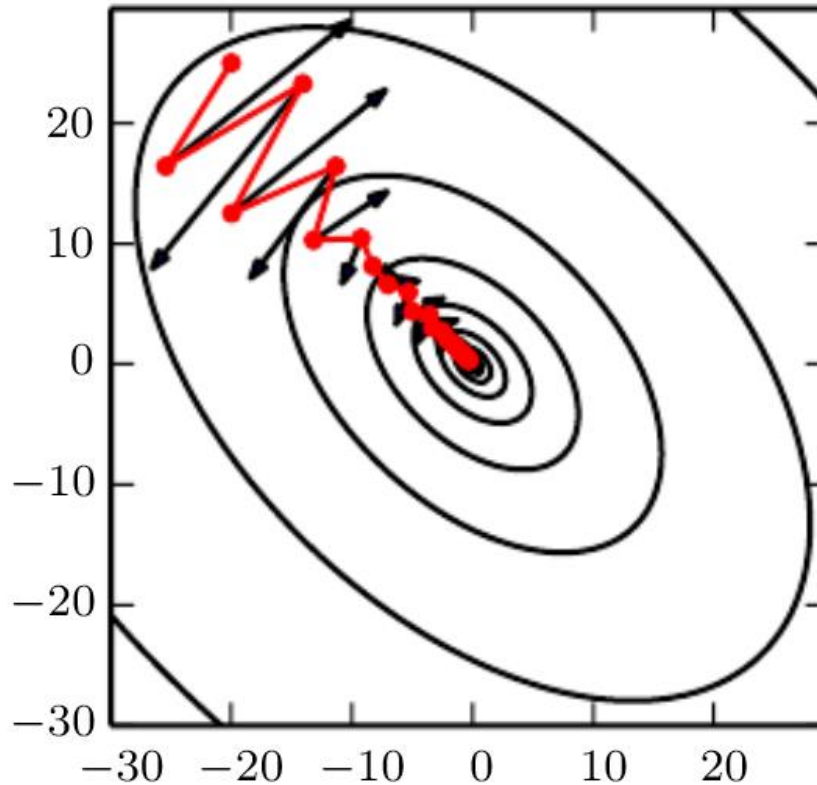


Illustration of how momentum traverses such an error surface better compared to Gradient Descent

# SGD with Momentum

---

## Algorithm 2 Stochastic Gradient Descent with Momentum

---

**Require:** Learning rate  $\epsilon_k$

**Require:** Momentum Parameter  $\alpha$

**Require:** Initial Parameter  $\theta$

**Require:** Initial Velocity  $\mathbf{v}$

1: **while** stopping criteria not met **do**

2: Sample example  $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$  from training set

3: Compute gradient estimate:

4:  $\hat{\mathbf{g}} \leftarrow +\nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

5: Compute the velocity update:

6:  $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$

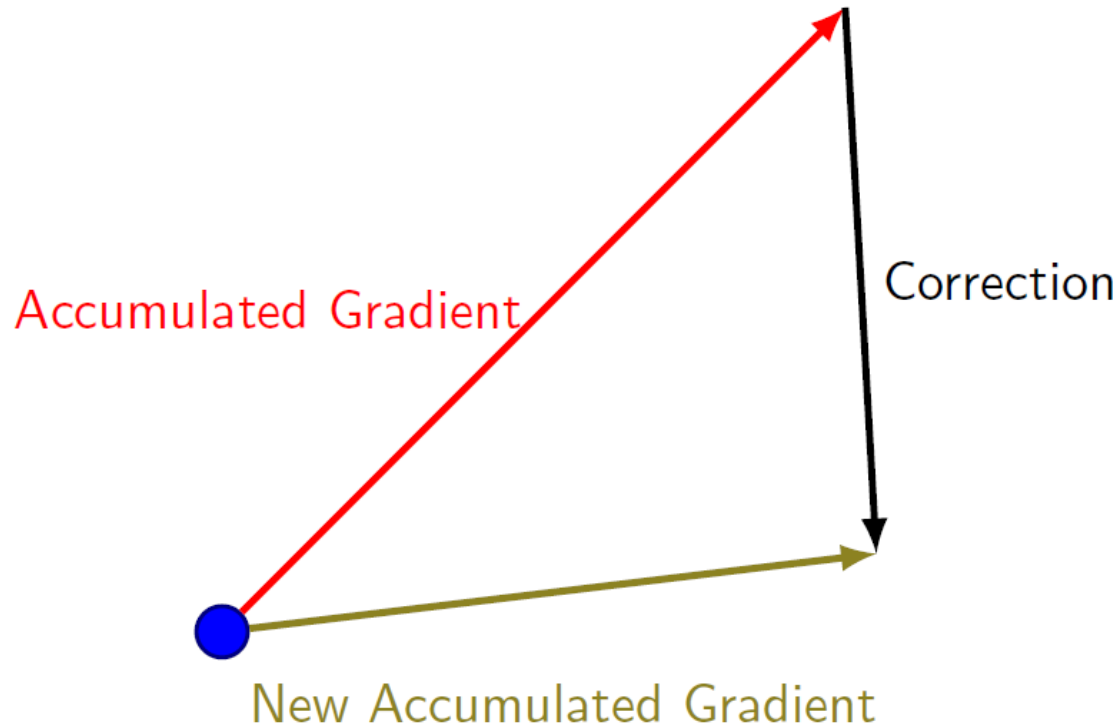
7: Apply Update:  $\theta \leftarrow \theta + \mathbf{v}$

8: **end while**

---

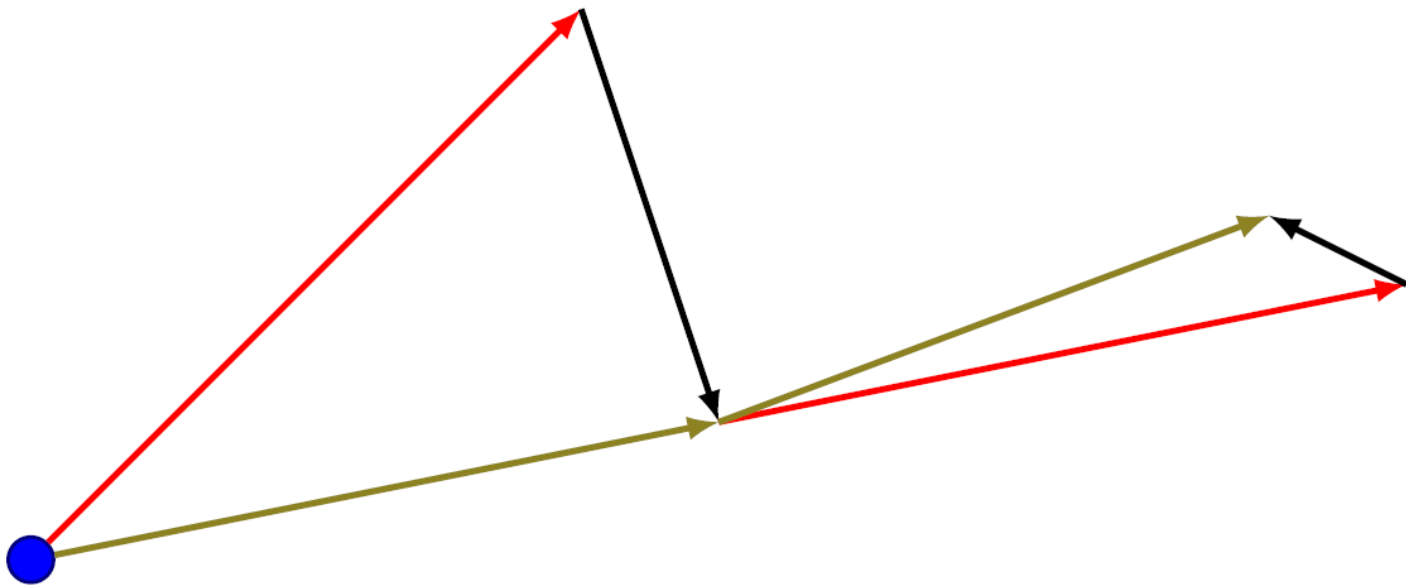
# Nesterov Momentum

- Another approach: First take a step in the direction of the accumulated gradient
- Then calculate the gradient and make a correction



# Nesterov Momentum

Next Step



# Nesterov Momentum

- Recall the velocity term in the Momentum method:

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

- Nesterov Momentum:

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

- Update:  $\theta \leftarrow \theta + \mathbf{v}$

# SGD with Nesterov Momentum

---

## Algorithm 3 SGD with Nesterov Momentum

---

**Require:** Learning rate  $\epsilon$

**Require:** Momentum Parameter  $\alpha$

**Require:** Initial Parameter  $\theta$

**Require:** Initial Velocity  $\mathbf{v}$

1: **while** stopping criteria not met **do**

2: Sample example  $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$  from training set

3: Update parameters:  $\tilde{\theta} \leftarrow \theta + \alpha \mathbf{v}$

4: Compute gradient estimate:

5:  $\hat{\mathbf{g}} \leftarrow +\nabla_{\tilde{\theta}} L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$

6: Compute the velocity update:  $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$

7: Apply Update:  $\theta \leftarrow \theta + \mathbf{v}$

8: **end while**

---

# Adaptive Learning Rate Methods

# Motivation

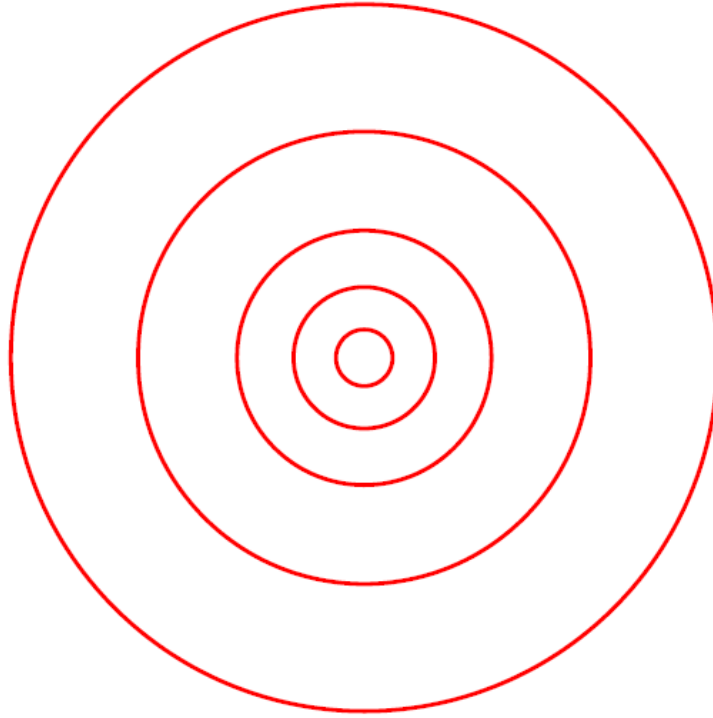


- Till now we assign the same learning rate to all parameters
- If the parameters vary in importance and frequency, why is this a good idea?
- It's probably not!



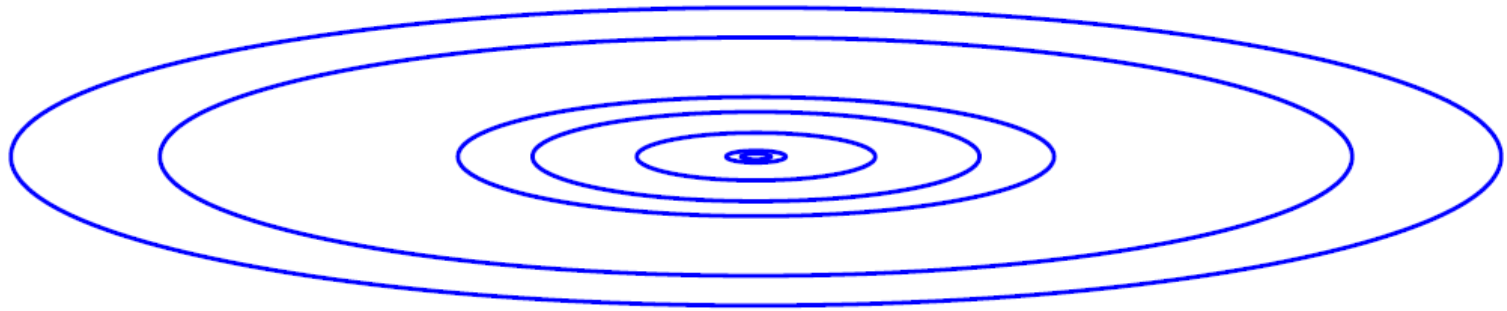
# Motivation

---



Nice (all parameters are equally important)

# Motivation



Harder!

# AdaGrad



- Idea: Downscale a model parameter by square-root of sum of squares of all its historical values
- Parameters that have large partial derivative of the loss - learning rates for them are rapidly declined
- Some interesting theoretical properties

# AdaGrad

---

## Algorithm 4 AdaGrad

---

**Require:** Global Learning rate  $\epsilon$  Initial Parameter  $\theta, \delta$

Initialize  $\mathbf{r} = 0$

1: **while** stopping criteria not met **do**

2: Sample example  $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$  from training set

3: Compute gradient estimate:  $\hat{\mathbf{g}} \leftarrow +\nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

4: Accumulate:  $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$

5: Compute update  $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$

6: Apply Update:  $\theta \leftarrow \theta + \Delta\theta$

8: **end while**

---

# RMSProp



- AdaGrad is good when the objective is convex.
- AdaGrad might shrink the learning rate too aggressively, we want to keep the history in mind
- We can adapt it to perform better in non-convex settings by accumulating an exponentially decaying average of the gradient
- This is an idea that we use again and again in Neural Networks
- Currently has about 500 citations on scholar, but was proposed in a slide in Geoffrey Hinton's coursera course

# RMSProp

---

## Algorithm 4 RMSProp

---

**Require:** Global Learning rate  $\epsilon$  , decay Parameter  $\rho$  ,  $\delta$

Initialize  $\mathbf{r} = 0$

1: **while** stopping criteria not met **do**

2: Sample example  $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$  from training set

3: Compute gradient estimate:  $\hat{\mathbf{g}} \leftarrow +\nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

4: Accumulate:  $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$

5: Compute update  $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$

6: Apply Update:  $\theta \leftarrow \theta + \Delta\theta$

8: **end while**

---

# RMSProp with Nesterov

---

## Algorithm 4 RMSProp with Nesterov

---

**Require:** Global Learning rate  $\epsilon$  , decay Parameter  $\rho$  ,  $\delta$  ,  $\alpha$ ,  $\mathbf{v}$

Initialize  $\mathbf{r} = 0$

1: **while** stopping criteria not met **do**

2: Sample example  $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$  from training set

3: Compute update:  $\tilde{\theta} \leftarrow \theta + \alpha \mathbf{v}$

4: Compute gradient estimate:  $\hat{\mathbf{g}} \leftarrow +\nabla_{\tilde{\theta}} L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$

5: Accumulate:  $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$

6: Compute Velocity:  $\mathbf{v} \leftarrow \alpha \mathbf{v} - \frac{\epsilon}{\sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$

7: Apply Update:  $\theta \leftarrow \theta + \mathbf{v}$

8: **end while**

---

# Adam

---

- We could have used RMSProp with momentum
- Use of Momentum with rescaling is not well motivated
- Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments



# Adam: Adaptive Moments

---

**Algorithm 8.7** The Adam algorithm

---

**Require:** Step size  $\epsilon$  (Suggested default: 0.001)

**Require:** Exponential decay rates for moment estimates,  $\rho_1$  and  $\rho_2$  in  $[0, 1)$ .  
(Suggested defaults: 0.9 and 0.999 respectively)

**Require:** Small constant  $\delta$  used for numerical stabilization. (Suggested default:  $10^{-8}$ )

**Require:** Initial parameters  $\theta$

Initialize 1st and 2nd moment variables  $\mathbf{s} = \mathbf{0}$ ,  $\mathbf{r} = \mathbf{0}$

Initialize time step  $t = 0$

**while** stopping criterion not met **do**

    Sample a minibatch of  $m$  examples from the training set  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ .

    Compute gradient:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

    Update biased first moment estimate:  $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$

    Update biased second moment estimate:  $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

    Correct bias in first moment:  $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$

    Correct bias in second moment:  $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$

    Compute update:  $\Delta \theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \delta}}$  (operations applied element-wise)

    Apply update:  $\theta \leftarrow \theta + \Delta \theta$

**end while**

---

# Comparison between These Methods

$$\text{SGD: } \theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$$

$$\text{Momentum: } \mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}} \text{ then } \theta \leftarrow \theta + \mathbf{v}$$

$$\text{Nesterov: } \mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\theta} \left( L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right) \text{ then } \theta \leftarrow \theta + \mathbf{v}$$

$$\text{AdaGrad: } \mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g} \text{ then } \Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g} \text{ then } \theta \leftarrow \theta + \Delta\theta$$

$$\text{RMSProp: } \mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g} \text{ then } \Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g} \text{ then } \theta \leftarrow \theta + \Delta\theta$$

$$\text{Adam: } \hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t} \text{ then } \Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta} \text{ then } \theta \leftarrow \theta + \Delta\theta$$

# Batch Normalization

- Method to reparameterize a deep network to reduce co-ordination of update across layers
- Can be applied to input layer, or any hidden layer
- Let  $H$  be a design matrix having activations in any layer for  $m$  examples in the mini-batch

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1k} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{m1} & h_{m2} & h_{m3} & \dots & h_{mk} \end{bmatrix}$$

# Batch Normalization

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1k} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{m1} & h_{m2} & h_{m3} & \dots & h_{mk} \end{bmatrix}$$

- Each row represents all the activations in layer for one example
- **Idea**: Replace  $H$  by  $H'$  such that:

$$H' = \frac{H - \mu}{\sigma}$$

- $\mu$  is mean of each unit and  $\sigma$  the standard deviation

# Batch Normalization

---

- $\mu$  is a vector with  $\mu_j$  the column mean
- $\sigma$  is a vector with  $\sigma_j$  the column standard deviation
- $H_{i,j}$  is normalized by subtracting  $\mu_j$  and dividing by  $\sigma_j$

# Batch Normalization

- During training we have:

$$\mu = \frac{1}{m} \sum_j H_{:,j}$$

$$\sigma = \sqrt{\delta + \frac{1}{m} \sum_j (H - \mu)_j^2}$$

- We then operate on  $H'$  as before  $\implies$  we backpropagate **through** the normalized activations

# An Innovation

---

- Standardizing the output of a unit can limit the expressive power of the neural network
- Solution: Instead of replacing  $H$  by  $H'$ , replace it with  $\gamma H' + \beta$
- $\gamma$  and  $\beta$  are also learned by backpropagation

# Batch Normalization

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

“Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” Ioffe and Szegedy 2015



# Why is this good?

---

- The update will never act to only increase the mean and standard deviation of any activation
- Previous approaches added penalties to cost or per layer to encourage units to have standardized outputs
- Batch normalization makes the reparameterization easier
- **At test time:** Use running averages of  $\mu$  and  $\sigma$  collected during training, use these for evaluating new input  $x$

# Batch Normalization: Conclusion

---

- Improves gradient flow through the network.
- Allows higher learning rates.
- Reduces the strong dependence on initialization.
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout.

# Initialization Strategies

---

- In convex problems with good  $\epsilon$  no matter what the initialization, convergence is guaranteed
- In the non-convex regime initialization is much more important
- Some parameter initialization can be unstable, not converge
- Neural Networks are not well understood to have principled, mathematically nice initialization strategies
- What is known: Initialization should break symmetry (quiz!)
- What is known: Scale of weights is important
- Most initialization strategies are based on intuitions and heuristics

# Some Heuristics

---

- For a fully connected layer with  $m$  inputs and  $n$  outputs, sample:

$$W_{ij} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

- Xavier Initialization: Sample

$$W_{ij} \sim U\left(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right)$$

- Xavier initialization is derived considering that the network
- consists of matrix multiplications with no nonlinearities
- Works well in practice!

# More Heuristics

---

- Saxe et al. 2013, recommend initializing to random orthogonal matrices, with a carefully chosen gain  $g$  that accounts for nonlinearities
- If  $g$  could be divined, it could solve the vanishing and exploding gradients problem (more later)
- The idea of choosing  $g$  and initializing weights accordingly is that we want norm of activations to increase, and pass back strong gradients
- Martens 2010, suggested an initialization that was sparse: Each unit could only receive  $k$  non-zero weights
- Motivation: It is a bad idea to have all initial weights to have the same standard deviation  $\frac{1}{\sqrt{m}}$

# Outline

---

- 1/ Course Review
- 2/ Regularization
- 3/ Optimization
- 4/ Project Grouping

# Project Logistics



- Final grade = 50% from the group project + 50% from the final exam
- All projects must be finished in a team and each team is composed of 5-10 members
- The group project contains a technical report and project presentation
- The project presentation is decided by random sampling due to time limit.

# Project Logistics



- The project presentation will be held in the last course. Each team selects one student to report their work and the reporting time is 5 to 10 minutes.
- The technical report must contain responsibility and workload of each member in addition to regular project description. At the same time, the final technical report should be attached with source code which can be reproduced as they present in the technical report.
- The technical report must be written in English with no page limitation and the latex template can be downloaded from [http://cvpr2018.thecvf.com/submission/main\\_conference/author\\_guidelines](http://cvpr2018.thecvf.com/submission/main_conference/author_guidelines) .
- Considering the difficulty degree of different tasks, we will give appropriate scores in the final grade.



# Project Logistics



Projects will be evaluated based on:

1. **The technical quality of the work.** (I.e., Does the technical material make sense? Are the things tried reasonable? Are the proposed algorithms or applications clever and interesting? Do the authors convey novel insight about the problem and/or algorithms?)
2. The **completeness** and **novelty** of the work, and the **clarity** of the write-up. (Spare enough time for the write-up since it may be harder than you imagine)

# Project Logistics

---

Your final report is expected to be a 4~8 page report. You should submit both an electronic and a hardcopy version for your final report. It should roughly have the following format:

- Introduction - Motivation
- Problem definition
- Proposed method
  - Intuition - why should it be better than other methods?
  - Description of its algorithms
- Experiments
  - Description of your testbed; list of questions your experiments are designed to answer
  - Details of the experiments; observations
- Conclusions

# 1.Human Action Recognition

---

**Project idea:** Suppose that we are given a few video clips. Can we classify whether these videos contain humans ? If yes, can we track the humans in the videos, detect the pose of human body parts, and eventually infer what activity the human make? In this project, you are encouraged **to implement your own human action recognition system**. Since this area of research is so vast, we do not recommend you to try a general problem. More specified is better. Here are some references as a starting point.

Following is a list of data sets you could use.

[1] KTH:<http://www.nada.kth.se/cvap/actions/>

[2] Weizmann:<http://www.wisdom.weizmann.ac.il/~vision/SpaceTimeActions.html>

[3] Hollywood Human Actions

dataset:<http://www.irisa.fr/vista/Equipe/People/Laptev/download.html>

[4] VIRAT Video Dataset:<http://www.viratdata.org/>

([http://groups.inf.ed.ac.uk/calvin/articulated\\_human\\_pose\\_estimation\\_code/](http://groups.inf.ed.ac.uk/calvin/articulated_human_pose_estimation_code/))

# 1. Human Action Recognition

---

Here are some references as a starting point.

- [1] CVPR 2011 Tutorial on Human Activity Recognition  
(<http://cvrc.ece.utexas.edu/mryoo/cvpr2011tutorial/>)
- [2] Human Activity Recognition Summer course  
(<http://www.cs.sfu.ca/~mori/courses/cmpt888/summer10/>)
- [2] Stanford Vision lab ([http://vision.stanford.edu/discrim\\_rf/](http://vision.stanford.edu/discrim_rf/))  
(<http://ai.stanford.edu/~bangpeng/ppmi.html>)
- [3] Poselet (<http://www.cs.berkeley.edu/~lbourdev/poselets/>)
- [4] 2D articulated human pose estimation

You could use any code from the web for computing spatial-temporal features. One good example is the spatial-temporal interest point proposed by Piotr Dollar. Source code available at <http://vision.ucsd.edu/~pdollar/research/research.html>.

---

# 2. Image Categorization

**Project idea:** Image categorization/object recognition has been one of the most important research problems in the computer vision community. Researchers have developed a wide spectrum of different local descriptors, feature coding schemes, and classification methods.

In this project, **you will implement your own object recognition system**. You could use any code from the web for computing image features, such as SIFT, HoG, etc.

For computing SIFT features, you could use <http://www.vlfeat.org/~vedaldi/code/sift.html>.

**You're also encouraged to try the state-of-the-art deep learning methods:**

<http://deeplearning.net/>

Following is a list of data sets you could use.

[1] Caltech101/256:

[http://www.vision.caltech.edu/Image\\_Datasets/Caltech101/Caltech101.html](http://www.vision.caltech.edu/Image_Datasets/Caltech101/Caltech101.html)

[2] The PASCAL Object Recognition Database Collection:

<http://pascallin.ecs.soton.ac.uk/challenges/VOC/databases.html>

[3] LabelMe: <http://labelme.csail.mit.edu/>

[4] Face in the wild: <http://vis-www.cs.umass.edu/lfw/>

[5] ImageNet: <http://www.image-net.org/index>

[6] TinyImage: <http://groups.csail.mit.edu/vision/TinyImages/>

# 3. Multi-modal Social Media

---

**Project idea:** In this project, we encourage students to infer the underlying relations between different modalities of information on the Web. Here are some examples.

(1) Given a photo of movie poster (image), can we retrieve related trailers (video) or latest news articles (documents) of the movie?

To make project simpler, we recommend focusing on less than five movies (eg. 'Rise of the planet of the apes' and 'The smurfs'). You first download posters and trailers from some well-organized sites such as [imdb.com](http://imdb.com) or [itunes.com](http://itunes.com). They will be used as training data to learn your classifiers. Now your job is to gather raw data from [youtube.com](http://youtube.com) or Flickr, and classify them. In this project, we encourage you to explore the possibility to build classifiers to be learned from one information modality (eg. images), and to be applicable to other modalities (eg. trailer videos).

---

# 3. Multi-modal Social Media

---

(2) Given a beer label (image), can we search for which frames of a given video clip the logo or bottle appears?

Suppose that you are a big fan of Guinness beer. You can easily download the clean Guinness logo or cup images by Google image search. These images can be used to learn your detector, which can discover the frames that the logo appears in the video clips. For testing, you can download some video clips from youtube.com.

The above examples are just two possible candidates, and any new ideas or problem definitions are welcome.

For this purpose, one may take advantage of some source codes available on the Web as unit modules (e.g., near-duplicated image detection, object recognition, action recognition in video).

Another interesting direction is to improve the current state-of-the-arts methods by considering more practical scenarios.

---

# 3. Multi-modal Social Media

## Related Papers and Software::

- A good example of how a machine learning technique is successfully applied to real systems (ex. Google news recommendation).

- [1] Das, Datar, Garg, Rajaram. Google news personalization: scalable online collaborative filtering. WWW 2007.

- One of most popular approaches to near duplicated image detection is LSH families.

- [2] <http://www.mit.edu/~andoni/LSH/> (This webpage links several introductory articles and source codes).

- Various hashing techniques in computer vision (papers and source codes).

- [3] Spectral Hashing (<http://www.cs.huji.ac.il/~yweiss/SpectralHashing/>)

- [4] Kernelized LSH (<http://www.eecs.berkeley.edu/~kulis/klsh/klsh.htm>)

- Recognition in video

- [5] Naming of Characters in Video (<http://www.robots.ox.ac.uk/~vgg/data/nface/index.html>)

- [6] Action recognition in Video

- (<http://www.robots.ox.ac.uk/~vgg/data/stickmen/index.html>)

- Recognition in images

- [7] Human pose detection (Poselet) (<http://www.eecs.berkeley.edu/~lbourdev/poselets/>)

- [8] General object detection (<http://people.cs.uchicago.edu/~pff/latent/>)



# Other Projects

---

4. Image Segmentation
5. Face Recognition
6. Text Classification
7. Question Answering
8. Image Denosing/Super-resolution
9. Image Retrieval
10. Tracking
11. .....

**Note that** not all topics are equally difficult. But we'll take this into consideration when evaluating each team's performance and make the assessment as fair as possible.

# Acknowledgement



Some of the materials in these slides are drawn inspiration from:

- Shubhendu Trivedi and Risi Kondor, University of Chicago, Deep Learning Course
- Hung-yi Lee, National Taiwan University, Machine Learning and having it Deep and Structured course
- Xiaogang Wang, The Chinese University of Hong Kong, Deep Learning Course
- Fei-Fei Li, Standord University, CS231n Convolutional Neural Networks for Visual Recognition course

# Next time

---

- Deep Generative Models

# Questions?

---

# Thank You !



WeChat Group for Deep Learning