



THE UNIVERSITY *of* EDINBURGH
informatics

Machine Learning Practical (MLP)

Deep Neural Networks (3)

Changjian Li

Prevent Overfitting

Regularisation: Penalizing Complexity

- Consider adding a *complexity term* E_w to the network error function, to encourage smoother mappings:

$$E^n = \underbrace{E_{\text{train}}^n}_{\text{data term}} + \underbrace{\lambda E_w}_{\text{prior term}}$$

Regularisation: Penalizing Complexity

- Consider adding a *complexity term* E_w to the network error function, to encourage smoother mappings:

$$E^n = \underbrace{E_{\text{train}}^n}_{\text{data term}} + \underbrace{\lambda E_W}_{\text{prior term}}$$

- E_{train} is the usual error function:

$$E_{\text{train}}^n = - \sum_{k=1}^K t_k^n \ln y_k^n$$

Regularisation: Penalizing Complexity

- Consider adding a *complexity term* E_w to the network error function, to encourage smoother mappings:

$$E^n = \underbrace{E_{\text{train}}^n}_{\text{data term}} + \underbrace{\lambda E_W}_{\text{prior term}}$$

- E_{train} is the usual error function:

$$E_{\text{train}}^n = - \sum_{k=1}^K t_k^n \ln y_k^n$$

- E_W should be a differentiable flexibility/complexity measure, e.g.

$$E_W = E_{L2} = \frac{1}{2} \sum_i w_i^2$$

$$\frac{\partial E_{L2}}{\partial w_i} = w_i$$

Gradient Descent Training with Weight Decay

$$\boxed{\frac{\partial E^n}{\partial w_i}} = \frac{\partial(E_{\text{train}}^n + E_{L2})}{\partial w_i} = \left(\frac{\partial E_{\text{train}}^n}{\partial w_i} + \lambda \frac{\partial E_{L2}}{\partial w_i} \right) = \left(\frac{\partial E_{\text{train}}^n}{\partial w_i} + \lambda w_i \right)$$

$$\Delta w_i = -\eta \left(\frac{\partial E_{\text{train}}^n}{\partial w_i} + \lambda w_i \right)$$

$$w_i = w_i - \eta \left(\frac{\partial E_{\text{train}}^n}{\partial w_i} + \lambda w_i \right) = (1.0 - \eta\lambda)w_i - \eta \frac{\partial E_{\text{train}}^n}{\partial w_i}$$

- Weight decay corresponds to adding $E_{L2} = 1/2 \sum_i w_i^2$ to the error function
- Addition of complexity terms is called *regularisation*
- When used with gradient descent, weight decay corresponds to L2 regularisation

L1 Regularisation

- **L1 Regularisation** corresponds to adding a term based on summing the absolute values of the weights to the error:

$$\begin{aligned} E^n &= \underbrace{E_{\text{train}}^n}_{\text{data term}} + \underbrace{\lambda E_{L1}^n}_{\text{prior term}} \\ &= E_{\text{train}}^n + \lambda \sum_i |w_i| \end{aligned}$$

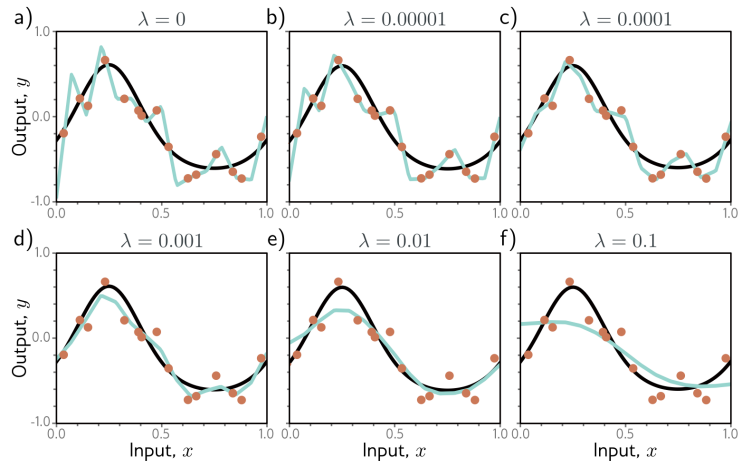
- Gradients

$$\begin{aligned} \frac{\partial E^n}{\partial w_i} &= \frac{\partial E_{\text{train}}^n}{\partial w_i} + \lambda \frac{\partial E_{L1}}{\partial w_i} \\ &= \frac{\partial E_{\text{train}}^n}{\partial w_i} + \lambda \operatorname{sgn}(w_i) \end{aligned} \quad \text{where} \quad \operatorname{sgn}(w_i) = \begin{cases} 1, & \text{if } w_i > 0 \\ -1, & \text{if } w_i < 0 \end{cases}$$

L1 vs L2

- L1 and L2 regularisation both have the effect of penalising larger weights
 - In L2 they shrink to 0 at a rate proportional to the size of the weight (λw_i)
 - In L1 they shrink to 0 at a constant rate ($\lambda \operatorname{sgn}(w_i)$)
- Behaviour of L1 and L2 regularisation with large and small weights:
 - when $|w_i|$ is large L2 shrinks faster than L1
 - when $|w_i|$ is small L1 shrinks faster than L2
- L1 tends to shrink some weights to be exact 0, leaving a few large important connections – L1 encourages *sparsity*, while L2 reduces all weights together
- $\partial E_{L1} / \partial w$ is undefined when $w = 0$; use the subgradient $[-1, 1]$ (*i.e.*, use 0 in the update equation)

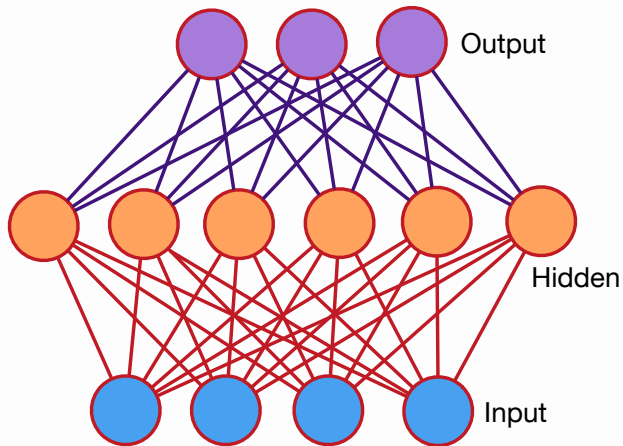
Regularisation and Capacity



Increasing the regularisation parameter λ reduces the effective model capacity

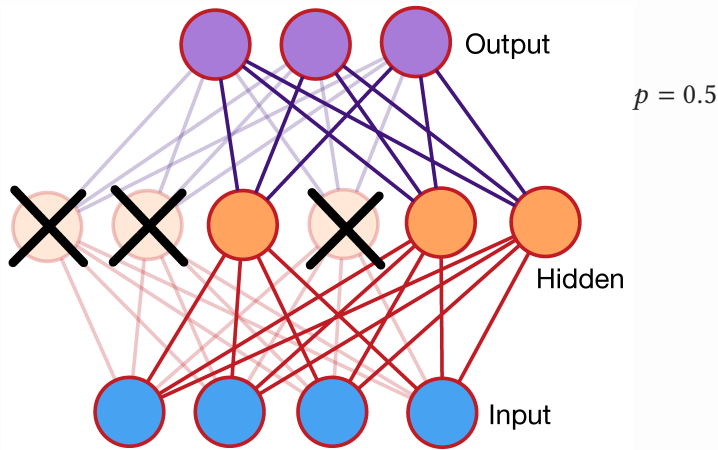
Regularisation: Dropout

Complete Network



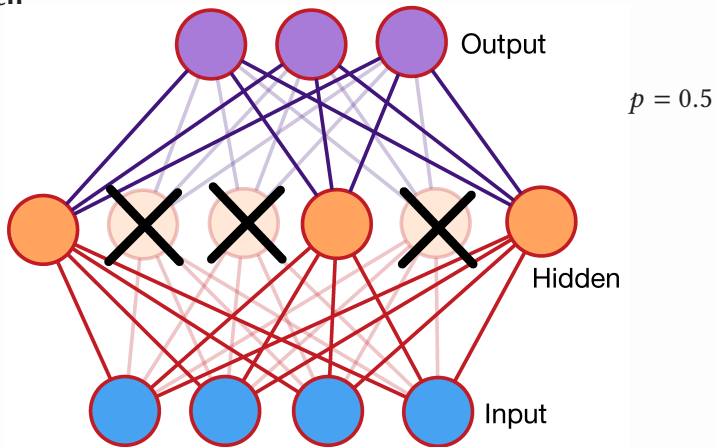
Regularisation: Dropout

First Minibatch



Regularisation: Dropout

Second Minibatch



Regularisation: Dropout

- **Dropout** is a way of training networks to behave so that they have the behaviour of an average of multiple networks
 - Each mini-batch randomly delete a fraction of the hidden units (inclusion probability $p \sim 0.5$) – and their related weights and biases
 - Then process the mini-batch (forward and backward) using this modified network, and update the weights
 - Restore the deleted units/weights, choose a new random subset of hidden units to delete and repeat the process

Regularisation: Dropout

- When training is complete the network will have learned a complete set of weights and biases, all learned when a fraction $(1 - p)$ of the hidden units are missing. To compensate for this, in the final network we scale hidden unit activations by p (i.e. the probability that a weight is included).
- To compensate for missing units, use “inverted dropout”: scale by $1/p$ when training, no scaling in final network (default).
- **Annealed Dropout:** Dropout rate schedule starting with a fraction $(1 - p)$ units dropped, decreasing at a constant rate to 0. (No longer common)
 - Initially training with dropout
 - Eventually fine-tune all weights together

What might Dropout be doing
to alleviate overfitting?

Why does Dropout work (intuitively)?

- Each mini-batch is like training a different network, since we randomly select to dropout (remove) a fraction of the units
- So we can imagine dropout as combining an exponential number of networks
- Since the component networks will be complementary and overfit in different ways, dropout is implicit model combination
- Also interpret dropout as training more robust hidden unit features – each hidden unit cannot rely on all other hidden unit features being present, must be robust to missing features
- Dropout has been useful in improving the generalisation of large-scale deep networks

Reduce Overfitting: Data Augmentation

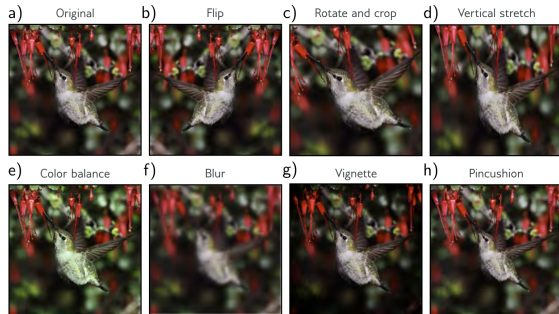


Figure 9.13 Data augmentation. For some problems, each data example can be transformed to augment the dataset. a) Original image. b–h) Various geometric and photometric transformations of this image. For image classification, all these images still have the same label, “bird.” Adapted from Wu et al. (2015a).

- Generalisation performance goes with the amount of training data
- Given a finite training set we could *create* further training examples...
 - Create new examples by making small rotations of existing data
 - Add a small amount of random noise
- Using “realistic” distortions to create new data is better than adding random noise

Other ways to regularise & improve generalisation

- Add noise to the input and target labels during training (label smoothing)
- Model combination – smooth together multiple networks
- Transfer learning – use a network trained on a related task as a starting point
- Multi-task learning – train a network to perform multiple tasks simultaneously
- Parameter sharing – e.g. convolutional networks

Learning Algorithms

How to set
the learning rate?

Weight Updates

- Let $d_i(t) = \partial E / \partial w_i(t)$ be the gradient of the error function E with respect to a weight w_i at update time t
- “Vanilla” gradient descent updates the weight along the negative gradient direction:

$$\Delta w_i(t) = -\eta d_i(t)$$

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

Hyperparameter η - *learning rate*

- Initialise η , and update as the training progresses (learning rate schedule)

Learning Rate Schedules

- Proofs of convergence for stochastic optimisation rely on a learning rate that reduces through time (as $1/t$) - Robbins and Munro (1951)
- Learning rate schedule – typically initial larger steps followed by smaller steps for fine tuning: Results in *faster convergence* and *better solutions*

Learning Rate Schedules

- Proofs of convergence for stochastic optimisation rely on a learning rate that reduces through time (as $1/t$) - Robbins and Munro (1951)
- Learning rate schedule – typically initial larger steps followed by smaller steps for fine tuning: Results in *faster convergence* and *better solutions*
- **Time-dependent** schedules

$$\Delta w_i(t) = -\eta(t)d_i(t)$$

- **Piecewise constant:** pre-determined η for each epoch
- **Exponential:** $\eta(t) = \eta(0) \exp(-t/r)$ ($r \sim$ training set size)
- **Reciprocal:** $\eta(t) = \eta(0)(1 + t/r)^{-c}$ ($c \sim 1$)
- **Cyclical** η – e.g. cosine annealing: Use multiple cycles, set $\eta = \eta(0)$ and decay it at each cycle, restart it

Learning Rate Schedules

- Proofs of convergence for stochastic optimisation rely on a learning rate that reduces through time (as $1/t$) - Robbins and Munro (1951)
- Learning rate schedule – typically initial larger steps followed by smaller steps for fine tuning: Results in *faster convergence* and *better solutions*
- **Time-dependent** schedules

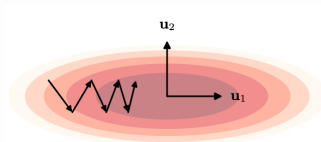
$$\Delta w_i(t) = -\eta(t)d_i(t)$$

- **Piecewise constant:** pre-determined η for each epoch
- **Exponential:** $\eta(t) = \eta(0) \exp(-t/r)$ ($r \sim$ training set size)
- **Reciprocal:** $\eta(t) = \eta(0)(1 + t/r)^{-c}$ ($c \sim 1$)
- **Cyclical** η – e.g. cosine annealing: Use multiple cycles, set $\eta = \eta(0)$ and decay it at each cycle, restart it
- **Performance-dependent** η – Fixed η until validation set stops improving, then divide η by 10 each epoch (i.e. constant, then exponential)

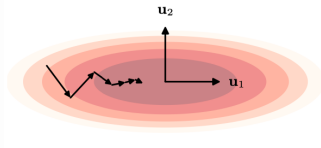
Training with Momentum

$$\Delta w_i(t) = -\eta d_i(t) + \alpha \Delta w_i(t-1)$$

- $\Delta w_i(t-1)$ is a moving average of previous gradients
- $\alpha \sim 0.9$ is the *momentum* hyperparameter
- Weight changes start by following the gradient
- After a few updates they start to have *velocity* – no longer pure gradient descent
- Momentum term encourages the weight change to go in the previous direction
- Damps the random directions of the gradients, to encourage weight changes in a consistent direction



Without momentum



With momentum

Reference: (Polyak, 1964)

Image credit: Fig 7.3,7.6,Bishop et al., DL

Adaptive Learning Rates

- Tuning learning rate (and momentum) parameters can be expensive (hyperparameter grid search) – it works, but we can do better
- Adaptive learning rates and per-weight learning rates
 - AdaGrad – normalise the update for each weight
 - RMSProp – AdaGrad forces the learning rate to always decrease, this constraint is relaxed with RMSProp
 - Adam – “RMSProp with momentum”

Well-explained by

Andrej Karpathy at <http://cs231n.github.io/neural-networks-3/>

Sebastian Ruder at <https://www.ruder.io/optimizing-gradient-descent/>

How should we set learning rate
when the gradient is big?

AdaGrad

- Separate, normalised update for each weight
- Normalised by the sum squared gradient S

$$S_i(0) = 0$$

$$S_i(t) = S_i(t-1) + d_i(t)^2$$

$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} d_i(t)$$

$\epsilon \sim 10^{-8}$ is a small constant to prevent division by 0 errors

AdaGrad

- Separate, normalised update for each weight
- Normalised by the sum squared gradient S

$$S_i(0) = 0$$

$$S_i(t) = S_i(t-1) + d_i(t)^2$$

$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} d_i(t)$$

- $\epsilon \sim 10^{-8}$ is a small constant to prevent division by 0 errors
- The update step for a parameter w_i is normalised by the (square root of) the sum squared gradients for that parameter
 - Weights with larger gradient magnitudes will have smaller effective learning rates
 - S_i cannot get smaller, so the effective learning rates monotonically decrease
- AdaGrad can decrease the effective learning rate too aggressively in deep networks

RMSProp

- RMSProp¹ using a moving average of the squared gradient – similar to AdaGrad, but replacing the sum by a moving average for S :

$$S_i(t) = \beta S_i(t-1) + (1 - \beta) d_i(t)^2$$
$$\Delta w_i(t) = \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} d_i(t)$$

$\beta \sim 0.9$ is the decay rate

RMSProp

- RMSProp¹ using a moving average of the squared gradient – similar to AdaGrad, but replacing the sum by a moving average for S :

$$S_i(t) = \beta S_i(t-1) + (1 - \beta) d_i(t)^2$$

$$\Delta w_i(t) = \boxed{\frac{-\eta}{\sqrt{S_i(t)} + \epsilon}} d_i(t)$$

$\beta \sim 0.9$ is the decay rate

- Effective learning rates no longer guaranteed to decrease

Adam

- Hinton commented about RMSProp: “Momentum does not help as much as it normally does”

Adam

- Hinton commented about RMSProp: “Momentum does not help as much as it normally does”
- Adam (Kingma & Ba, <https://arxiv.org/abs/1412.6980>) can be viewed as addressing this – it is a variant of RMSProp with momentum:

$$\begin{aligned}M_i(t) &= \alpha M_i(t-1) + (1-\alpha) d_i(t) \\S_i(t) &= \beta S_i(t-1) + (1-\beta) d_i(t)^2 \\ \Delta w_i(t) &= \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} M_i(t)\end{aligned}$$

Here a momentum-smoothed gradient is used for the update in place of the gradient. Kingma and Ba recommend $\alpha \sim 0.9$, $\beta \sim 0.999$

Adam

- Hinton commented about RMSProp: “Momentum does not help as much as it normally does”
- Adam (Kingma & Ba, <https://arxiv.org/abs/1412.6980>) can be viewed as addressing this – it is a variant of RMSProp with momentum:

$$\begin{aligned}M_i(t) &= \alpha M_i(t-1) + (1-\alpha)d_i(t) \\S_i(t) &= \beta S_i(t-1) + (1-\beta)d_i(t)^2 \\ \Delta w_i(t) &= \frac{-\eta}{\sqrt{S_i(t)} + \epsilon} M_i(t)\end{aligned}$$

Here a momentum-smoothed gradient is used for the update in place of the gradient. Kingma and Ba recommend $\alpha \sim 0.9$, $\beta \sim 0.999$

Summary

- Preventing overfitting (L1, L2, data augmentation)
- Dropout: learning more diverse features
- Learning rate schedules and adaptive learning algorithms
 - Heuristic
 - Adaptive
- **Recommended reading:**
 - Goodfellow et al, 7.12, 8.3, 8.5
 - A nice blog from Andrej Karpathy, CS231n notes (Stanford)
- **For keen readers:** Sebastian Ruder: An overview of gradient descent optimization



<https://forms.office.com/e/m8JJmxE6hQ>