Lecture Notes IV – Neural Networks, Part 2

Marina Meilă
mmp@uwaterloo.ca

With Thanks to Pascal Poupart & Gautam Kamath
Cheriton School of Computer Science
University of Waterloo

February 8, 2026

Training a single unit

Training a 2-layer network

Training a L-layer network

Backpropagation in practice

Practical remedies to I, T, S, L, O

Reading HTF Ch.: 11.3 Neural networks, Murphy Ch.: (16.5 neural nets), Bach Ch.: –, Deep Learning Book (Goodfellow, Bengio, Courville) 6.1-4, ResNet 7.6, ConvNet 9., Autoencoders 14.1, Dive Into Deep Learning 4.1-4.3.

Training a neural network

- Model is multilayer network.
- Layers $l = 1, 2, \ldots L$, with $x^{(l)} \in \mathbb{R}^{m_l}$, for $l = 1 : L-1$, $m_0 = d$, $m_L = 1$ (for regression and binary classification).

$$x^{(0)} = x \tag{1}$$
$$x^{(L)} = f(x) = z^{(L)} \equiv \phi_{\text{out}}(z^{(L)})^1 \tag{2}$$
$$x^{(l)} = \phi(z^{(l)}) \quad \text{for } l = 1 : L-1 \tag{3}$$
$$z^{(l)} = W^{(l)} x^{(l-1)} \tag{4}$$

- Parameters $\mathbb{W} = \{W^{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}\}$
- $W_k^{(l)}$ is row k of $W^{(l)}$ and corresponds to unit k of layer l
- $\mathcal{D} = \{(x^1, y^1), \ldots (x^n, y^n)\}$

- **How to train this network?**
- Minimize $\mathcal{L}(\mathbb{W})$ (remember ϕ_{out} is associated with $\mathcal{L}$)

- Minimization by Gradient Descent
  - Initialize $\mathbb{W}$ to small random values
  - for $t = 1, 2, \ldots$ do $\mathbb{W}^{t+1} \leftarrow \mathbb{W}^t - \eta \frac{\partial \mathcal{L}}{\partial \mathbb{W}}(\mathbb{W}^t)$

- **We need to compute gradient** $\frac{\partial \mathcal{L}(\mathbb{W})}{\partial W_{ij}^{(l)}}$ for all $i = 1 : m_l$, $j = 1 : m_{l-1}$, $l = 1 : L$.

[1] See discussion of ϕ_{out} on next page

Training a single unit

- The function $f(x; \mathbb{W}) = \phi_{\text{out}} \underbrace{\left(\sum_{j=1}^{d} w_j x_j \right)}_{z}$ with parameters $\mathbb{W} = w \in \mathbb{R}^d$.
- The loss function $\mathcal{L}(y, f(x; \mathbb{W})) =$ Least Squares, or Logistic (Max Likelihood) binary or multiclass.
  Fact For each $\mathcal{L}$, there exists a mapping $y \to y_*$, $z \to \phi_{\text{out}}(z)$ such that
  $\boxed{-\frac{\partial \mathcal{L}(y, f(x;\mathbb{W}))}{\partial f} = y_* - \phi_{\text{out}}} = \boxed{\text{target output} - \text{network output}}$

Training a single unit

- The function $f(x; \mathbb{W}) = \phi_{\text{out}} \underbrace{\left(\sum_{j=1}^{d} w_j x_j \right)}_{z}$ with parameters $\mathbb{W} = w \in \mathbb{R}^d$.
- The loss function $\mathcal{L}(y, f(x; \mathbb{W})) =$ Least Squares, or Logistic (Max Likelihood) binary or multiclass.
  Fact For each $\mathcal{L}$, there exists a mapping $y \to y_*$, $z \to \phi_{\text{out}}(z)$ such that
  $\boxed{-\frac{\partial \mathcal{L}(y, f(x;\mathbb{W}))}{\partial f} = y_* - \phi_{\text{out}}} = \boxed{\text{target output} - \text{network output}}$
- Let's use chain rule

$$
\begin{aligned}
-\frac{\partial \mathcal{L}}{\partial f} &= y_* - \phi_{\text{out}}(z) && (5)\\
\frac{\partial f}{\partial z} &= \phi'_{\text{out}}(z) && (6)\\
\frac{\partial f}{\partial w} &= \phi'_{\text{out}}(z)x && (7)\\
\frac{\partial f}{\partial x} &= \phi'_{\text{out}}(z)w && (8)\\
-\frac{\partial \mathcal{L}}{\partial w} &= (y_* - \phi_{\text{out}}(z))\phi'_{\text{out}}(z)x
\end{aligned}
$$

- Remember $\phi'_{\text{out}}(z) = z$ for $\mathcal{L}_{LS}$, $\phi'_{\text{out}}(z) = \phi(x)(1 - \phi(z))$ for $\mathcal{L}_{\text{logit}}$.
- Notice that $\partial \mathcal{L}/\partial w$ is a vector collinear with x.

Training a single unit, n data points

- $\mathcal{D} = \{(x^1, y^1), \ldots (x^n, y^n)\}$
- For single pair (x, y), $\frac{\partial L}{\partial w} = (y - \phi_{\text{out}}(z))\phi'_{\text{out}}(z)x$
- For entire $\mathcal{D}$, $\mathcal{L}(\mathbb{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(y^i, f(x^i; \mathbb{W}))$.
- The gradient of this loss is

$$-\frac{\partial L}{\partial w} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial}{\partial w}\mathcal{L}(y^i, f(x^i; \mathbb{W})) = \frac{1}{n}\sum_{i=1}^{n}(y^i - \phi_{\text{out}}(z^i))\phi'_{\text{out}}(z^i)x^i. \quad (9)$$

Training a 2-layer network

- Consider a two layer neural network

$$f(x) = \phi_{\text{out}}\left(\overbrace{\sum_{i=1}^{m} w_i x_i^{(1)}}^{z^{(2)}}\right) = \phi_{\text{out}}\left(\sum_{i=1}^{m} w_i \overbrace{\phi(\sum_{j=1}^{d} w_{ij} x_j)}^{x^{(1)}}\right) \tag{10}$$

The parameters $\mathbb{W}$ are w and $W = [w_{ij}]_{i=1:m,j=1:d}$

Gradient w.r.t. w

$$-\frac{\partial \mathcal{L}}{\partial f} = y_* - \phi_{\text{out}}(z^{(2)}) \tag{11}$$

$$\frac{\partial f}{\partial x^{(1)}} = \phi'_{\text{out}}(z^{(2)})w \tag{12}$$

$$\frac{\partial f}{\partial x_i^{(1)}} = \phi'_{\text{out}}(z^{(2)})w_i \quad \text{(same as above, for single } i = 1:m) \tag{13}$$

$$\frac{\partial f}{\partial w} = \phi'_{\text{out}}(z^{(2)})x^{(1)} \tag{14}$$

$$-\frac{\partial \mathcal{L}}{\partial w} = (y_* - \phi_{\text{out}}(z^{(2)}))\phi'_{\text{out}}(z^{(2)})x^{(1)} \tag{15}$$

Trainining a 2-layer network – hidden layer parameters

Gradient w.r.t. W

- Let's break $W \in \mathbb{R}^{m \times d}$ into rows. $W_i = [w_{ij}]_{j=1:d}$ is the row vector of weights for unit i in hidden layer.
- We have $z_i = W_i^T x$ and $x_i^{(1)} = \phi(z_i)$.
- This is the same as for the single layer network, with $\phi_{\text{out}} \leftarrow \phi$, $w \leftarrow W_i$, $z \leftarrow z_i$.

$$\frac{\partial x_i^{(1)}}{\partial z_i} = \phi'(z_i) \tag{16}$$

$$\frac{\partial x_i^{(1)}}{\partial W_i} = \phi'(z_i)x \tag{17}$$

Recall from the output layer $z^{(2)} = w^T x_i^{(1)}$, $f = \phi_{\text{out}}(z^{(2)})$.

$$\frac{\partial f}{\partial W_i} = \frac{\partial f}{\partial x_i^{(1)}} \frac{\partial x_i^{(1)}}{\partial W_i} = \phi'_{\text{out}}(z^{(2)}) w_i \phi'(z_i) x \tag{18}$$

$$-\frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial \mathcal{L}}{\partial f} \frac{\partial f}{\partial W_i} = (y_* - \phi_{\text{out}}(z^{(2)})) \phi'_{\text{out}}(z^{(2)}) w_i \phi'(z_i) x \tag{19}$$

Training a 2-layer network – Computational savings

- when $f(x^i)$ is computed, $z_j(x^i)$ are too; they should be "cached" and re-used
- the derivative of ϕ is easily obtained from the ϕ value
- Exercise The above gradient formulas can be easilty written in matrix-vector form

Backpropagation extends recursively to multi-layer networks.

From 2 layers to L layers

$$\frac{\partial x_i^{(l)}}{\partial z_i^{(l)}} = \phi'(z_i^{(l)}) \tag{20}$$

$$\frac{\partial x_i^{(l)}}{\partial W_i^{(l)}} = \phi'(z_i^{(l)}) x^{(l-1)} \tag{21}$$

$$\frac{\partial x_i^{(l)}}{\partial x_i^{(l-1)}} = \phi'(z_i^{(l)}) W_i^{(l)} \tag{22}$$

$$\frac{\partial f}{\partial x_i^{(l)}} = \sum_{j=1}^{m_{l+1}} \frac{\partial f}{\partial x_j^{(l+1)}} \frac{\partial x_j^{(l+1)}}{\partial x_i^{(l)}} \tag{23}$$

$$\frac{\partial f}{\partial W_i^{(l)}} = \frac{\partial f}{\partial x_i^{(l)}} \frac{\partial x_i^{(l)}}{\partial W_i^{(l)}} \tag{24}$$

$$-\frac{\partial \mathcal{L}}{\partial W_i^{(l)}} = (y_* - \phi_{\text{out}}) \frac{\partial f}{\partial W_i^{(l)}} \tag{25}$$

Practical properties of backpropagation

- Unlike in logistic regression, $\mathcal{L}$ has many local optima even for two layers and simple problems.
- Hence, initialization is important, and there are no general rules for a good initialization. Even if the neural network works well, we do not know if we are at the optimum.
- Saturation If $\tilde{z}_j = w_j^T x$ is large in magnitude, then $z_j = \phi(\tilde{z}_j)$ is near 0 or 1. In either case, $\phi'(\tilde{z}_j) = z_j(1 - z_j) \approx 0$. We say that that this sigmoid is **saturated**; z_j will be virtually insensitive to changes in w_j[2]
  To avoid saturation at the beginning of the training, one initializes W with "small" (w.r.t $\max ||x^i||$, random values. Exercise Why random and not exactly 0?
- To speed up training, it is useful to standardize the input data[3] $x^{1:N}$ as a preprocessing step. Exercise Note that theoretically shifting and rescaling the data should NOT have any effect.
- J can have **plateaus**, i.e. regions where $\nabla J \approx 0$ but that do not contain a local minimum. Exercise What can cause plateaus? Exercise And what is bad about them?

- In conclusion, training neural networks by backpropagation is an art: requires experience with the algorithm, careful tuning, repeated restarts, and a long time.

[2] or to changes in previous layers, if this is a multilayer network.

[3] Standardization should NOT include the dummy coordinate $x^0 \equiv 1$.

Backpropagation – some issues

- I Computation – how many ops / iteration?
- T Convergence – how many iterations (T) ?
- S Saturation – $\phi'(x) \approx 0$ for large $|z|$
- L Local minima
- 0 Overfitting?

Ops/ iteration

- Number parameters p
  - $\mathbb{W} = \{W^{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}, \, l = 1 : L\}$
  - Hence $p \sim \sum_{l=1}^{L} m_{l-1} m_l$
- Forward pass: each $W_{ij}^{(l)}$ is used once to propagate from $x_j^{(l-1)}$ to $x_i^{(l)}$, for each data point
- Backward pass: each $W_{ij}^{(l)}$ is used once to propagate from $x_i^{(l)}$ to $x_j^{(l-1)}$, for each data point
- Update: each weight is updated once.
- Hence number operations / iteration is $\mathcal{O}(np)$

- This can become very large with deep networks, large data, and large input dimensions $d \, (= m_0)$

T Convergence – how many iterations?

- **Theory** – Gradient Descent (GD) is "order 1", a slower method (asymptotically) compared to Newton.
  - 1-st order optimization method: uses only 1-st derivative info
  - 2-nd order optimization method: uses 1-st and 2-nd derivative info
  - 0-th order optimization method: uses no derivative info (only $\mathcal{L}$ values)
- In the case of neural networks, as for any $\mathcal{L}$ with local minima, the practical issues below are also relevant.

- Progress of training is slow when the gradient $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)}}$ is ≈ 0 away from the optimum
- When does this happen? $\mathcal{L}$ has valleys (in some direction $\frac{\partial \mathcal{L}}{\partial \mathbb{W}} \approx 0$) or plateaus, or saddles
- Saturation

L Local optima

- $\mathcal{L}$ has **many local optima** and valleys (often between local optima) [and plateaus]
- It is not possible or expected to find the global optimum in $\mathbb{W}$ space (note also there are multiple equal optima)
- The goal of training is
  - to avoid the really bad local optima
  - to avoid stopping at saddles or plateaus
- Modern (very large) nn
  - have extremely many local minima
  - have extremely many good local minima
  - GD (and SGD) converge very close to the starting point!!
  - . . . and GD favors local minima with **good generalization** !!

Overfitting

- p can be very large deep networks
- Overfitting possible for everyday/in-house/not too large networks
- (Remedies later on in this lecture)

- Modern (very large) nn regime
  - Very large nn, with $p \gg n$ **are not** subject to the classical bias-variance tradeoff
  - **This was a surprising discovery!** The phenomenon is called **double descent**
  - (A separate lecture on it if time)

Stochastic gradient descent – I,T,L

SGD

1. At each iteration t, select a **batch** $\mathcal{B} \subset \mathcal{D}$ of size $n' \ll n$ from the data
2. Calculate gradient and update only with respect to the samples in $\mathcal{B}$

$$\mathbb{W}^{t+1} \leftarrow \mathbb{W}^t - \eta \frac{1}{n'} \sum_{i \in \mathcal{B}} \frac{\partial \mathcal{L}(y^i, f(x^i))}{\partial \mathbb{W}}(\mathbb{W}^t) \tag{26}$$

Stochastic gradient descent – I,T,L

SGD

1. At each iteration t, select a **batch** $\mathcal{B} \subset \mathcal{D}$ of size $n' \ll n$ from the data
2. Calculate gradient and update only with respect to the samples in $\mathcal{B}$

$$\mathbb{W}^{t+1} \leftarrow \mathbb{W}^t - \eta \frac{1}{n'} \sum_{i \in \mathcal{B}} \frac{\partial \mathcal{L}(y^i, f(x^i))}{\partial \mathbb{W}}(\mathbb{W}^t) \tag{26}$$

- What is achieved?
  - Faster gradient calculation! $\sim n'p$ instead of $\sim np$
  - $g_{\mathcal{B}} \approx g_{\mathcal{D}}$ where $g_{\mathcal{B},\mathcal{D}}$ represent the gradients on the batch, respectively entire $\mathcal{D}$
  - If $\mathcal{B}$ is a random subset, then $g_{\mathcal{B}}$ is a random variable with mean $g_{\mathcal{D}}$.
- Variations and refinements
  - use $n' = 1$ possible (update weights after every data point)
  - start with n' small (to advance fast) then increase it (to reduce "noise")
  - naturally adapts to on-line learning (use $n' \geq 1$ data points, then discard them)
- Randomness in gradient can help avoid very poor local minimima, saddles, etc (not plateaus)
- Impact on **I**: faster iteration, **T**: arrrive faster near the local minimumum, **L**: as above
- Terminology: GD also called batch GD, SGD with $n' > 1$ also called minibatch SGD, some people understand by SGD the SGD with $n' = 1$ (typical in algorithm analysis); epoch a pass through the entire $\mathcal{D}$, e.g. n/n' iterations of SGD.

Can we mimic 2-nd order methods "cheaply"? (T,L)

- "2-nd derivative" is Hessian $\frac{\partial^2 \mathcal{L}}{\partial \mathbb{W}^2}$ a $p \times p$ matrix
- We want to get the benefits of 2-nd order in $\mathcal{O}(p)$ time.[4]

- Let $g \in \mathbb{R}^p$ denote a gradient or stochastic gradient (p is still the number of parameters we are training)
- **Momentum (Heavy ball method)**

$$\mathbb{W}^{t+1} \leftarrow \mathbb{W}^t - \gamma\eta g^t + (1-\gamma)\underbrace{(\mathbb{W}^t - \mathbb{W}^{t-1})}_{\text{previous step}} \tag{27}$$

- (many variations exist, e.g. Nesterov method)

- Adaptive learning rates (coming next)

[4] The factor n or n' is ignored, because it does not affect what we do.

Adaptive Learning Rates (LR) – **S,T**

- Typical for SGD
- The LR adapts over iteration t and parameter w_i
- Goal η "normalized" with a factor that avoids large updates

- $g_i^t = \frac{\partial \mathcal{L}}{\partial w_i}(\mathbb{W}^t)$
- Let $G_{i,t} = \sum_{t=1}^{t} (g_i^{t'})^2 =$ sum of squares of gradients for weight i
- AdaGrad algorithm

$$w_i^{t+1} \leftarrow w_i^t - \frac{\eta}{\sqrt{G_{i,t} + \epsilon}} g_i^t. \tag{28}$$

- Problem stepsize can't increase if needed

Adaptive Learning Rates (LR) – **S,T**

- Typical for SGD
- The LR adapts over iteration t and parameter w_i
- Goal η "normalized" with a factor that avoids large updates

- $g_i^t = \frac{\partial \mathcal{L}}{\partial w_i}(\mathbb{W}^t)$
- Let $G_{i,t} = \sum_{t=1}^{t} (g_i^{t'})^2 =$ sum of squares of gradients for weight i
- AdaGrad algorithm

$$w_i^{t+1} \leftarrow w_i^t - \frac{\eta}{\sqrt{G_{i,t} + \epsilon}} g_i^t. \tag{28}$$

- Problem stepsize can't increase if needed
- RMSProp "momentum" (forgetting) $G_{i,t} = 0.9 G_{i,t-1} + 0.1 (g_i^t)^2$

Adaptive Learning Rates (LR) – S,T

- Typical for SGD
- The LR adapts over iteration t and parameter w_i
- Goal η "normalized" with a factor that avoids large updates

- $g_i^t = \frac{\partial \mathcal{L}}{\partial w_i}(\mathbb{W}^t)$
- Let $G_{i,t} = \sum_{t=1}^{t} (g_i^{t'})^2$ = sum of squares of gradients for weight i
- AdaGrad algorithm

$$w_i^{t+1} \leftarrow w_i^t - \frac{\eta}{\sqrt{G_{i,t} + \epsilon}} g_i^t. \tag{28}$$

- Problem stepsize can't increase if needed
- RMSProp "momentum" (forgetting) $G_{i,t} = 0.9 G_{i,t-1} + 0.1 (g_i^t)^2$
- Adam momentum and RMSProp
  - Hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

$$m_i^t = \beta_1 m_i^{t-1} + (1-\beta_1) g_i^t \quad \text{momentum for } g \tag{29}$$

$$v_i^t = \beta_2 v_i^{t-1} + (1-\beta_2)(g_i^t)^2 \quad \text{RMSProp, momementum for } g^2 \tag{30}$$

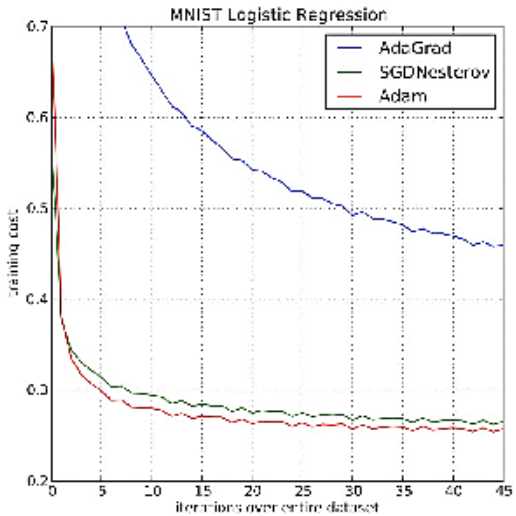
$$\hat{m}_i^t = \frac{m_i^t}{1-\beta_1^t} \quad \hat{v}_i^t = \frac{v_i^t}{1-\beta_2^t} \quad \text{decay} \tag{31}$$

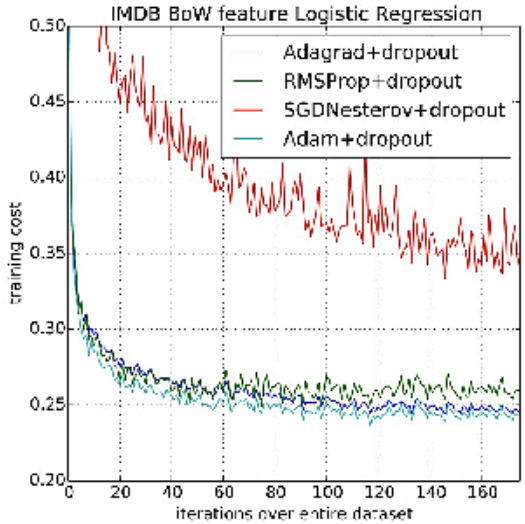
$$w_i^t \leftarrow w_i^{t-1} - \frac{\eta}{\sqrt{\hat{v}_i^t + \epsilon}} \hat{m}_i^t \tag{32}$$

Comparisons – AdaGrad, RMSProp, ADAM

Empirical Comparison

- From Kingma & Ba (ICLR-2015):





Normalizations – Controlling saturation **S,O**

- These refer to data/inputs to neurons, not to weights
- **Classic standardization**

$$\text{Recenter data around mean} \quad x^i \leftarrow x^i - \mu \qquad \mu = \frac{1}{n}\sum_{i=1}^{n} x^i \tag{33}$$

$$\text{Rescale each data coordinate} \quad x_j^i \leftarrow x_j^i/\sigma_j \qquad \sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n}(x_j^i)^2 \tag{34}$$

- Batch normalization – standardization within a batch
- Layer normalization – standardization within a layer

Batch normalization

- Normalize the inputs in each $\mathcal{B}$
- Normalize the inputs to the sigmoids in each batch and layer $\{z^{i(l)}, i \in \mathcal{B}\}$
- Normalize the inputs to the sigmoids in each batch and neuron $\{z_j^{i(l)}, i \in \mathcal{B}\}$, $j = 1 : m_l$, $l = 1 : L-1$
- Normalize the inputs to each layer, i.e. $x^{i(l)}$ for $x^i \in \mathcal{B}$, $l = 0 : L-1$

- Then add learnable scale and shift (same for all batches), e.g. $x^i \leftarrow \sigma_0 \frac{x^i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} + \mu_o$

Layer normalization

- Standardize $\{z_j^{(l)}, j = 1 : m_l\}$ for each data point and each layer

Avoiding overfitting O

- Regularization, e.g. weight decay
- Dropout
- Data augmentation
- Early stopping
- Model averaging (also called **bagging**) or Bayesian Neural Networks

Weight decay – O

- **Idea** Prevent overfitting by penalizing weights that are too large (Regularization)
- New "loss"

$$\mathcal{L}_\lambda(\mathbb{W}) = \mathcal{L}(\mathbb{W}) + \frac{\lambda}{2}\|\mathbb{W}\|^2 \tag{35}$$

and $\|\mathbb{W}\|^2 = \sum_{j=1}^{p} w_j^2$

- Effect on gradient $g_\lambda = \frac{\partial \mathcal{L}_\lambda}{\partial \mathbb{W}}$

$$\frac{\partial \mathcal{L}_\lambda}{\partial w_j} = \frac{\partial}{\partial w_j}\left(\mathcal{L}(\mathbb{W}) + \frac{\lambda}{2}\|\mathbb{W}\|^2\right) = \frac{\partial \mathcal{L}(\mathbb{W})}{\partial w_j} + \lambda w_j \tag{36}$$

$$w_j^{t+1} \leftarrow w_j^t - \eta\left(\frac{\partial \mathcal{L}(\mathbb{W})}{\partial w_j} + \lambda w_j^t\right) = w_j^t - \eta\frac{\partial \mathcal{L}(\mathbb{W})}{\partial w_j} - \eta\lambda w_j^t \tag{37}$$

$$= w_j^t(1-\eta\lambda) - \eta\frac{\partial \mathcal{L}(\mathbb{W})}{\partial w_j} \tag{38}$$

Dropout

Dropout

- Idea: randomly "drop" some units from the network when training
- Training: at each iteration of gradient descent
  - Each input unit is dropped with probability p_1 (e.g., 0.2)
  - Each hidden unit is dropped with probability p_2 (e.g., 0.5)
- Prediction (testing):
  - Multiply each input unit by $1 - p_1$
  - Multiply each hidden unit by $1 - p_2$

CS480/680 Winter 2023 - Lecture 11 - Pascal Poupart PAGE 20

UNIVERSITY OF WATERLOO

`https://www.cs.toronto.edu/ rsalakhu/papers/srivastava14a.pdf` "Dropout: A simple way to prevent neural networks from overfitting" by Srivastava, Hinton, Krizhevsky, Sutskever, Salkhutdinov, JMLR 2014.

Training with dropout

Train

- For each iteration
  - For each training example (x^i, y^i)
    1. Sample $r^{(l)} \in \{0,1\}^{m_l}$ from Bernoulli$(p^{(l)}_{\text{drop}})$, for $l = 1 : L$
    2. Backward propagation: skip/ignore dropped out units (do not compute gradient contribution from them)
    3. Update: only $w^{(l)}_i$ with $r^{(l)}_i = 1$

Test/Predict

- Forward propagation $x^{(l)} = \phi(W^{(l)} z^{(l)} (1 - p^{(l)}_{\text{drop}})$

Intuition

- Dropout can be viewed as an approximate form of ensemble learning

- In each training iteration, a different subnetwork is trained
- At test time, these subnetworks are “merged” by averaging their weights

Data augmentation – O

Early stopping