Lecture 4

Andrei Arsene Simion

Columbia University

February 9, 2023

Outline

- Neural Networks
- Optimization - Theory + Examples
- BackProp - Theory + Examples
- Residual Connections
- BatchNorm and LayerNorm
- Regularization - Dropout
- Wave Net

Outline

- Neural Networks
- Optimization - Theory + Examples
- BackProp - Theory + Examples
- Residual Connections
- BatchNorm and LayerNorm
- Regularization - Dropout
- Wave Net

Outline

- What is a neural network?
- Is it just a bunch of layers one after another?
- We apply 3 linear transformations to x

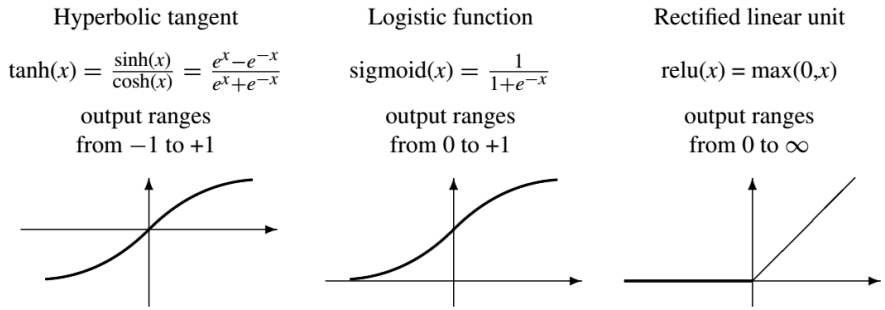
Linear Maps

- Apply the linear maps in sequence: this is equivalent to 1 linear map!
- $W_1 x + b_1 \sim Wx + b$
- $W_2(W_1 x + b_1) + b_2 = W_2 W_1 x + W_2 b_1 + b_2 \sim Wx + b$
- $W_3(W_2(W_1 x + b_1) + b_2) + b_3 =$
  $W_3 W_2 W_1 x + W_3 W_2 b_1 + W_3 b_2 + b_3 \sim Wx + b$
- Thus, if we just apply linear maps, it's like applying 1 (possibly big) linear map
- We **need** the nonlinearity between layers so that we get feature interactions that are more than linear in the parameter space

Nonlinearities

- Some famous nonlinearities:

Figure 5.3 Typical activation functions in neural networks.



- It is not clear which nonlinearity is best, but ReLU / GeLU is very popular
- There is active research on this
- Note that on the extremes of sigmoid and tanh the gradients are 0 - this is why they are not that popular anymore!
- When a gradient is 0 like this, we say that the activation is "saturated" ... We'll look later at the implications of this ...

XOR: What can a NN learn?

- Consider the XOR problem on x_0 and x_1
- True relationship:
  $x_0 \oplus x_1 = (x_0 \text{ OR } x_1) - (x_0 \text{ AND } x_1) \sim (x_0 + x_1) - x_0 * x_1$
- One network might be like below, where $h = \sigma(W_1 x + b_1)$
- The output: $y = W_2 h + b_2$

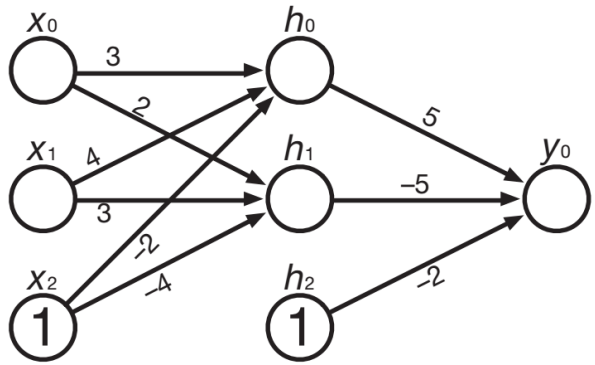


Figure 5.4 A simple neural network with bias nodes in input and hidden layers.

XOR

- We see h_0 is like OR, h_1 is like AND

Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.119	0.018	$0.183 \rightarrow 0$
0	1	0.881	0.269	$0.743 \rightarrow 1$
1	0	0.731	0.119	$0.743 \rightarrow 1$
1	1	0.993	0.731	$0.334 \rightarrow 0$

- For input (1, 0), we get 1 which is what we expect

Table 5.1 Calculations for Input (1,0) to the Network shown in Figure 5.4.

Layer	Node	Summation	Activation
Hidden	h_0	$1 \times 3 + 0 \times 4 + 1 \times -2 = 1$	0.731
Hidden	h_1	$1 \times 2 + 0 \times 3 + 1 \times -4 = -2$	0.119
Output	y_0	$0.731 \times 5 + 0.119 \times -5 + 1 \times -2 = 1.060$	0.743

- **It would not be possible to construct a network like this without the nonlinearities!**
- Try it!

Outline

- Neural Networks
- Optimization - Theory + Examples
- BackProp - Theory + Examples
- Regularization - Dropout
- BatchNorm and LayerNorm
- Residual Connections
- Wave Net

Optimization

- We usually have a problem like $\min_\theta(L(\theta))$ where L is some loss function
- L usually depends on your problem, and on the training data
  - For example: $L(\theta) = |y - \theta|$ or $L(\theta) = \frac{\sum_{i=1}^{n}(y_i - \theta_0 - \theta_1 x_i)^2}{n}$
- More complicated examples: Skip-Gram, CBOW, GloVe
- There are recipes we use to optimize these problems

Optimization

- An example of gradient descent

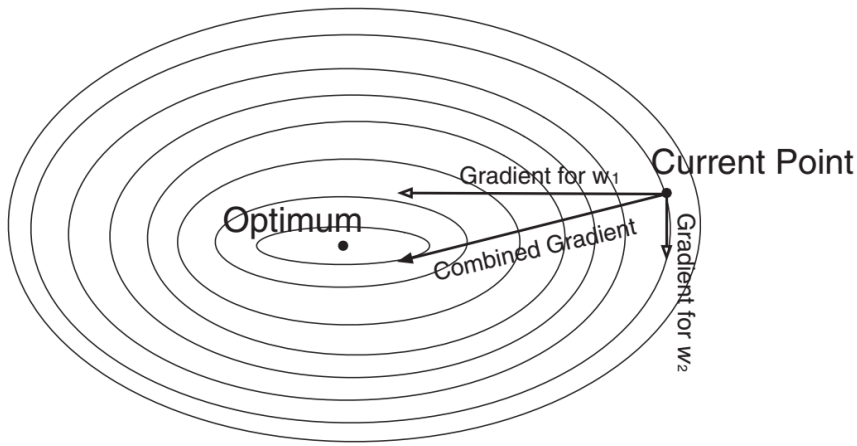


Figure 5.5 Gradient descent training. We compute the gradient in regard to every dimension. In this case the gradient with respect to weight w_2 is smaller than the gradient with respect to the weight w_1, so we move more to the left than down (note: arrows point in negative gradient direction, pointing to the minimum).

SGD

- Stochastic Gradient Descent is the most popular method to optimize $\min_\theta(L(\theta))$
- Divide your training set into batches $B_1, \ldots, B_T$ which cover your entire data
- Compute the gradient of L on these batches (i.e. assume the full data is just batch B_t and consider L_{B_t} and optimize over this set)
- Update the parameters: $\theta_t = \theta_{t-1} - \mu \frac{\partial L_{B_t}}{\partial \theta}(\theta_{t-1})$
- We compute $\frac{\partial L_B}{\partial \theta}$ over batches and this is like a proxy for the (expensive, full) $\frac{\partial L}{\partial \theta}$

SGD

- SGD vs Gradient Descent
- In case 2, Gradient Descent goes to a bad place
- Idea: Randomization helps!

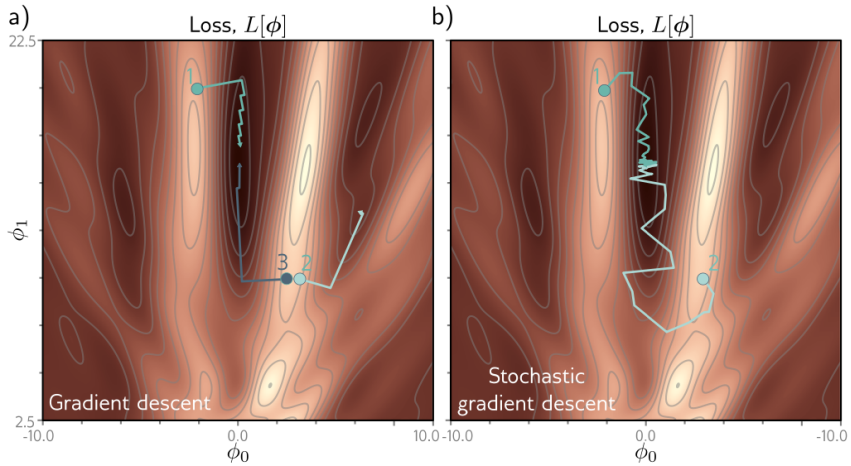


Figure 6.5 Gradient descent vs. stochastic gradient descent. a) Gradient descent with line search. As long as the gradient descent algorithm is initialized in the right "valley" of the loss function (e.g., points 1 and 3), the parameter estimate will move steadily toward the global minimum. However, if it is initialized outside this valley (e.g., point 2), it will descend toward one of the local minima. b) Stochastic gradient descent adds noise to the optimization process, so it is possible to escape from the wrong valley (e.g., point 2) and still reach the global minimum.

SGD: Batch size

- Batches can be of size 1, 256, 128, etc.
- The smaller the batch, the noisier the estimate
- But, the smaller the batch, the more gradient updates (learning) you can do
- Since we compute $\frac{\partial L_B}{\partial \theta}$ over batches, this is a noisy estimate for the gradient of $\frac{\partial L}{\partial \theta}$ over the entire training set
- What are some methods that deal with this noise?

Momentum

- We keep track of a momentum term
  $m_t = \beta m_{t-1} + (1-\beta)\frac{\partial L_{B_t}}{\partial \theta}(\theta_{t-1})$
- The idea is $\frac{\partial L_{B_t}}{\partial \theta}$ is choppy, but m_t is smoother
- The update would now be, for batch t, update via
  $\theta_t = \theta_{t-1} - \mu m_t$
- As we go forward with iterations, we put less and less weight on older gradient estimates, but we never forget old gradients

Momentum

- Momentum adds smoothness to the loss trajectory
- The gradient estimate has "less variance"

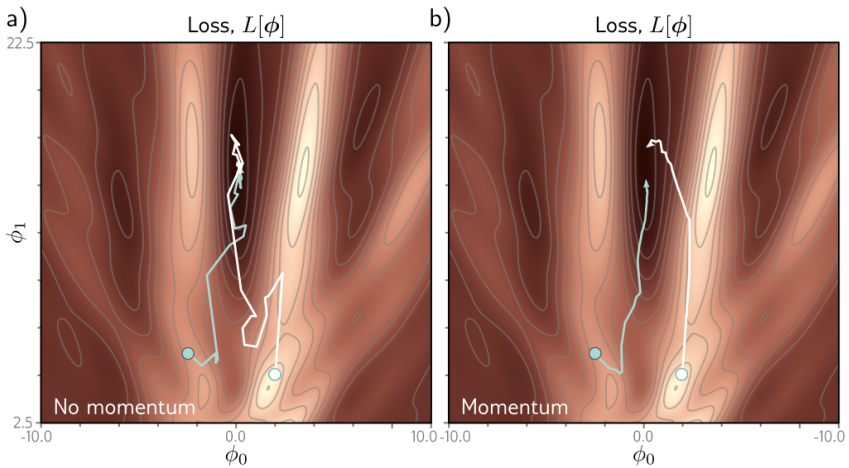


Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.

Nesterov Momentum

- The momentum term can be considered a coarse prediction of where the SGD algorithm will move next
- Nesterov accelerated momentum computes the gradient at this point rather than at the current point
- $m_t = \beta m_{t-1} + (1-\beta)\frac{\partial L_{B_t}}{\partial \theta}(\theta_{t-1} - \mu m_{t-1})$
- $\theta_t = \theta_{t-1} - \mu m_t$
- One way to think about this is that the gradient term now corrects the path provided by momentum alone

Nesterov Momentum

- Nesterov Momentum tries to pick the direction in a better way

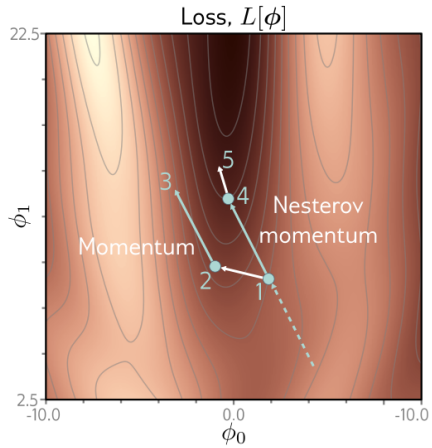


Figure 6.8 Nesterov accelerated momentum. The solution has traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same direction as the dashed line), arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4) and then measures the gradient and applies an update to arrive at point 5.

Issues with SGD

- Gradient descent with a fixed step size has the following undesirable property: it makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious) and small adjustments to parameters associated with small gradients (where perhaps we should explore further)
- When the gradient of the loss surface is much steeper in one direction than another, it is dificult to choose a learning rate that (i) makes good progress in both directions and (ii) is stable (figures 6.9a–b)

Adam

- Straightforward approach: is to normalize the gradients so that we move a fixed distance (governed by the learning rate) in each direction
- $m_t = \frac{\partial L_{B_t}}{\partial \theta}(\theta_{t-1})$
- $v_t = (\frac{\partial L_{B_t}}{\partial \theta}(\theta_{t-1}))^2$
- The term v_t is the squared gradient, and the positive root of this is used to normalize the gradient itself, so all that remains is the sign in each coordinate direction
- $\theta_t = \theta_{t-1} - \mu \frac{m_t}{\sqrt{v_t}+\epsilon}$
- The result is that the algorithm moves a fixed distance μ along each coordinate, where the direction is determined by whichever way is downhill (see figures 6.9c)
- This simple algorithm makes good progress in both directions
- But, it will not converge unless it happens to land exactly at the minimum
- Instead, it will bounce back and forth around the minimum

Adam

- Adam goes a step further, we also consider variance's momentum
- $m_t = (\beta m_{t-1} + (1-\beta)\frac{\partial L_{B_t}}{\partial \theta}(\theta_{t-1}))/(1-\beta^t)$
- We divide above by $(1-\beta^t)$ to have better estimates at the start, but this effect goes away as t gets larger
- $v_t = (\gamma v_{t-1} + (1-\gamma)(\frac{\partial L_{B_t}}{\partial \theta}(\theta_{t-1}))^2)/(1-\gamma^t)$
- $\theta_t = \theta_{t-1} - \mu \frac{m_t}{\sqrt{v_t}+\epsilon}$
- The gradient magnitudes of neural network parameters can depend on their depth in the network
- Adam helps compensate for this tendency and balances out changes across the different layers
- In practice, Adam also has the advantage of being less sensitive to the initial learning rate because it avoids situations like those in figures 6.9a–b, so it doesn't need complex learning rate schedules

Adam

- Adam vs other methods

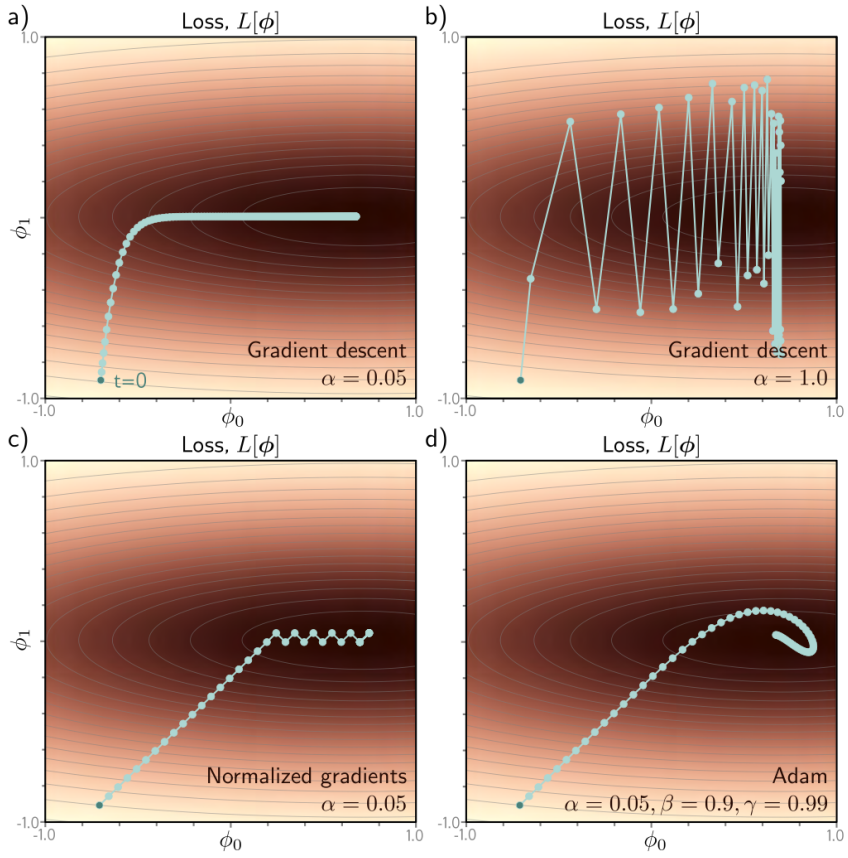


Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, it overshoots in the vertical direction and becomes unstable. c) A straightforward approach is to move a fixed distance along each axis at each step so that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude and retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term, which creates a smoother path.

Outline

- Neural Networks
- Optimization - Theory + Examples
- BackProp - Theory + Examples
- Residual Connections
- BatchNorm and LayerNorm
- Regularization - Dropout
- Wave Net

A basic NN

- Consider the formula given by the recursions below

$$
\begin{aligned}
s &= W_1 x + b_1 \\
h &= \text{sigmoid}(s) \\
z &= W_2 h + b_2 \\
y &= \text{sigmoid}(z)
\end{aligned}
$$

- Here, we take x as input, do a linear map of it through W_1 and b_1
- Apply a sigmoid to the result to get h
- Run the result of the sigmoid application through another linear map W_2 and b_2
- Finally, run this through another sigmoid to get the final result y

A basic NN

- Suppose the loss is the $L2$, Mean Square Error loss so that we are trying to minimize $e = 1/2(t - y)^2$
- We are interested in finding the optimal W_1, b_1, W_2 and b_2 for this loss
- We'll need $\frac{\partial e}{\partial W_1}$, $\frac{\partial e}{\partial W_2}$, $\frac{\partial e}{\partial b_1}$, $\frac{\partial e}{\partial b_2}$ at each iteration
- For SGD, the update rules are $\theta_{new} = \theta_{old} - \mu \frac{\partial e}{\partial \theta}(\theta_{old})$ where θ is any of the above 4 parameters

Chain Rule

- Recall the chain rule
- Let $A(\theta) = f(g(\theta))$
- Then,

$$A'(\theta) = f'(g(\theta))g'(\theta)$$

- Let $B(\theta) = f(g(h(\theta)))$
- Then,

$$B'(\theta) = f'(g(h(\theta)))g'(h(\theta))h'(\theta)$$

- *Basically, start at the end result (y) and traverse a graph of dependencies until you get from local derivatives the final derivative (W_1, b_2, etc.)*

Our NN

- For us, the graph might look like below
- Notice the leaf nodes!
- This is a DAG = Directed Acyclic Graph

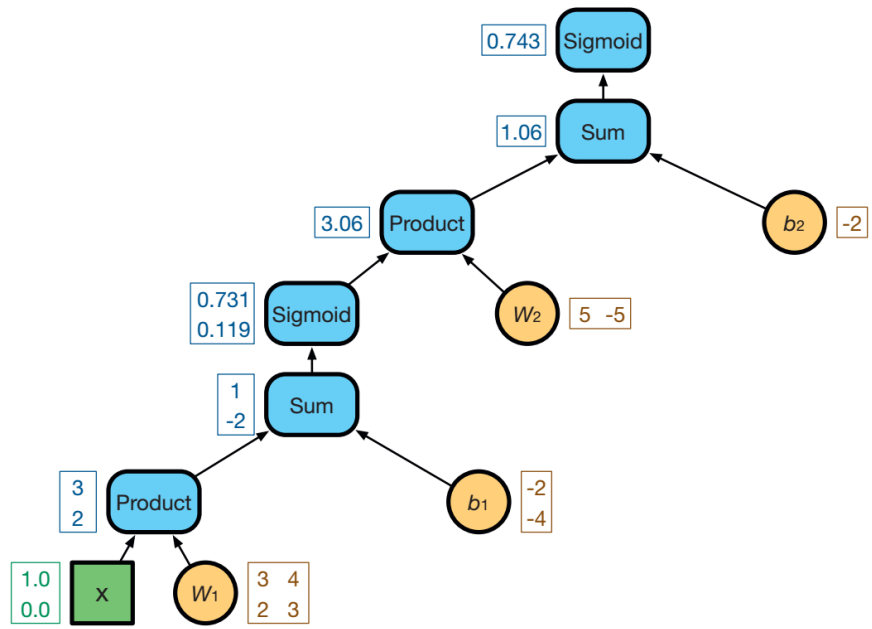


Figure 6.1 Two-layer feed-forward neural network as a computation graph, consisting of the input value x; weight parameters W_1, W_2, b_1, b_2; and computation nodes (product, sum, sigmoid). To the right of each parameter node, its value is shown. To the left of input and computation nodes, how the input $(1,0)^T$ is processed by the graph is shown.

Our NN

- Flushed out, this is:

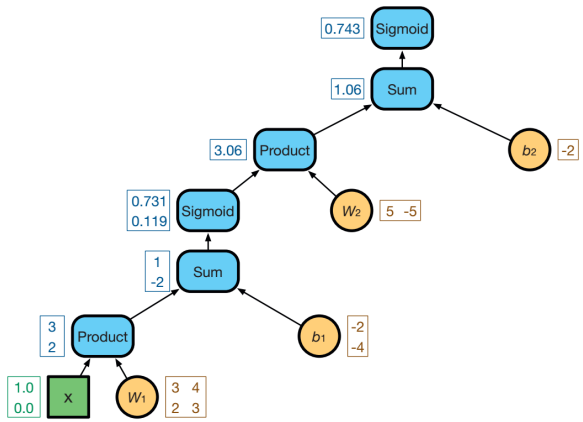
$$
\begin{aligned}
Product &= W_1 x \\
Sum &= Product + b_1 \\
Sigmoid &= sigmoid(Sum) \\
Product &= W_2 Sigmoid \\
Sum &= Product + b_2 \\
Sigmoid &= sigmoid(Sum)
\end{aligned}
$$

- Here, we have $y = Sigmoid$ and $e = \frac{(t-y)^2}{2}$ (not shown)
- What is $\frac{\partial e}{\partial W_2}$?

Our NN

- First compute for x and fixed θ the values of all intermediate representations
- This is Forward - Propagation
- Back - Propagation: start from the Loss, and go down to ALL paths to the final parameter leaf nodes
- Use the chain rule to break up the derivations into local derivatives that you know

Figure 6.1 Two-layer feed-forward neural network as a computation graph, consisting of the input value x; weight parameters W_1, W_2, b_1, b_2; and computation nodes (product, sum, sigmoid). To the right of each parameter node, its value is shown. To the left of input and computation nodes, how the input $(1,0)^T$ is processed by the graph is shown.



Our NN

- It is better to write the last few computations as

$$p = W_2 h$$
$$s = p + b_2$$
$$y = sigmoid(s)$$
$$e = MSE(y, t)$$

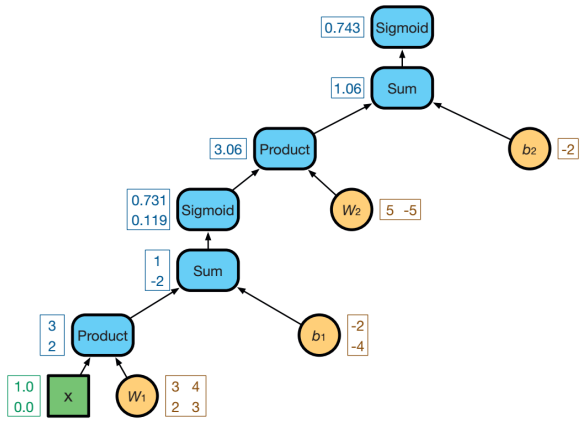


Figure 6.1 Two-layer feed-forward neural network as a computation graph, consisting of the input value x; weight parameters W_1, W_2, b_1, b_2; and computation nodes (product, sum, sigmoid). To the right of each parameter node, its value is shown. To the left of input and computation nodes, how the input $(1,0)^T$ is processed by the graph is shown.

Our NN

- We have
- $\frac{\partial e}{\partial W_2}$
- $\frac{\partial e}{\partial y}\frac{\partial y}{\partial W_2}$
- $\frac{\partial e}{\partial y}\frac{\partial y}{\partial s}\frac{\partial s}{\partial W_2}$
- $\frac{\partial e}{\partial y}\frac{\partial y}{\partial s}\frac{\partial s}{\partial p}\frac{\partial p}{\partial W_2}$

$$p = W_2 h$$
$$s = p + b_2$$
$$y = sigmoid(s)$$
$$e = MSE(y, t)$$

Our NN

- We need $\frac{\partial e}{\partial W_2} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial s}\frac{\partial s}{\partial p}\frac{\partial p}{\partial W_2}$

$$p = W_2 h$$
$$s = p + b_1$$
$$y = \sigma(s)$$
$$e = L2(y, t) = \frac{(t-y)^2}{2}$$

- This means $\frac{\partial e}{\partial y} = y - t$
- We have $\frac{\partial y}{\partial s} = \sigma(s)(1 - \sigma(s))$
- We have $\frac{\partial s}{\partial p} = 1$
- We have $\frac{\partial p}{\partial W_2} = h$
- So, we have the answer is $\frac{\partial e}{\partial W_2} = (y - t)\sigma(s)(1 - \sigma(s))h$
- We *have all these terms due to the forward pass*

Gradient Updates

- The derivatives we need can be written in the graph as below
- Note that we have in general $\theta_{new} = \theta_{old} - \mu \frac{\partial L}{\partial \theta}(\theta_{old})$

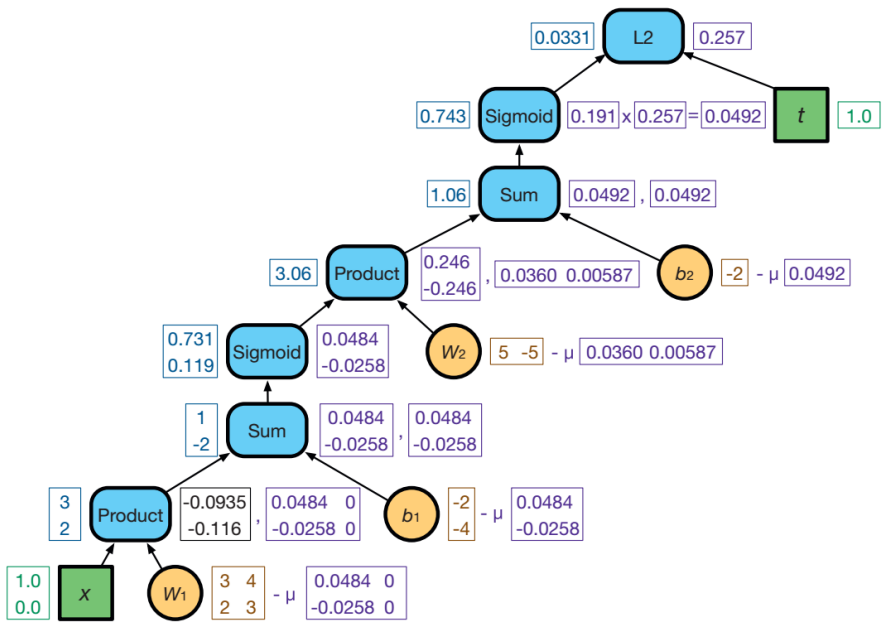


Figure 6.2 Computation graph with gradients computed in the backward pass for the training example $(0,1)^T \rightarrow 1.0$. Gradients are computed with respect to the input of the nodes, so some nodes that have two inputs also have two gradients. See text for details on the computations of the values.

PyTorch

- Effectively, PyTorch builds a graph like below and uses its structure to get recursions
- A gradient can be approximated numerically, for example $f'(x) \sim \frac{f(x-\epsilon)+f(x+\epsilon)}{2\epsilon}$
- There are better ways, but this is the idea

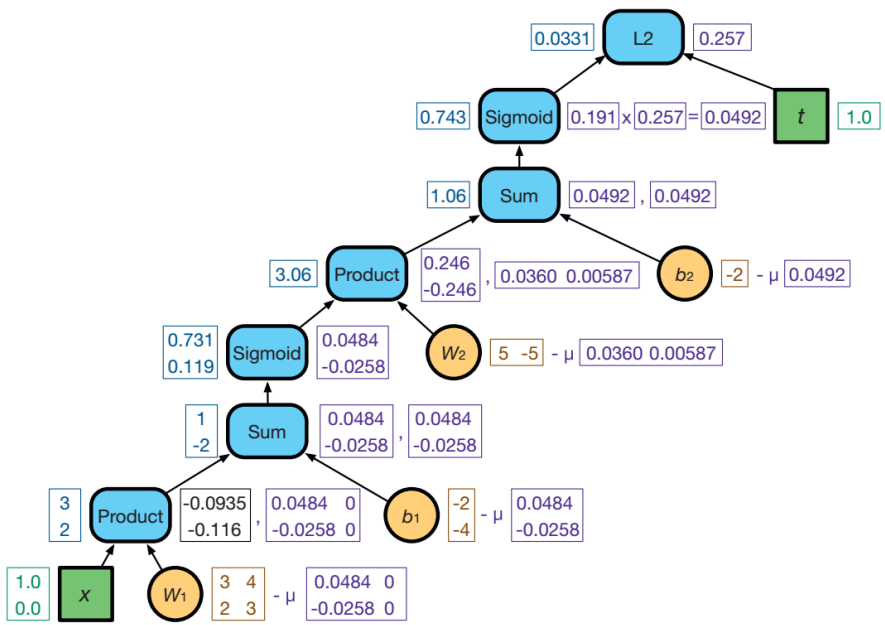


Figure 6.2 Computation graph with gradients computed in the backward pass for the training example $(0,1)^T \rightarrow 1.0$. Gradients are computed with respect to the input of the nodes, so some nodes that have two inputs also have two gradients. See text for details on the computations of the values.

Outline

- Neural Networks
- Optimization - Theory + Examples
- BackProp - Theory + Examples
- Residual Connections
- BatchNorm and LayerNorm
- Regularization - Dropout
- Wave Net

Vanishing Gradients

- In the last example, we had $\frac{\partial e}{\partial W_2} = (y - t)\sigma(s)(1 - \sigma(s))h$
- What happens if s is very large or very small?

What is s is small / large?

- If s is small, $\sigma(s) = 0$
- If s is large, $\sigma(s) = 1$
- In either case, we have $\sigma(s)(1 - \sigma(s)) = 0$ so that $\frac{\partial e}{\partial W_2} = 0$
- This is bad, since now there is no learning for W_2

Residual Connections

- What if we modify the last few computations to be

$$
\begin{aligned}
p &= W_2 h \\
s &= p + b_1 \\
y &= \sigma(s) + \mathbf{p} \\
e &= L2(y, t) = \frac{(t-y)^2}{2}
\end{aligned}
$$

- Then we have $\frac{\partial e}{\partial W_2} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial s}\frac{\partial s}{\partial p}\frac{\partial p}{\partial W_2} + \frac{\partial e}{\partial y}\frac{\partial y}{\partial p}\frac{\partial p}{\partial W_2}$
- Why? We can go $e \to y \to s \to p \to W_2$ or $e \to y \to p \to W_2$
- Even if s is large, the term $\frac{\partial e}{\partial y}\frac{\partial y}{\partial p}\frac{\partial p}{\partial W_2} = (y-t)(1)(h)$ will contribute

Residual Connections

- What if we modify the last few computations to be

$$
\begin{aligned}
p &= W_2 h \\
s &= p + b_1 \\
y &= \sigma(s) + \mathbf{s} \\
e &= L2(y, t) = \frac{(t - y)^2}{2}
\end{aligned}
$$

- Another way: $y = \sigma(s) + \mathbf{s}$
- But are $\mathbf{s}$ (or $\mathbf{p}$) and $\sigma(s)$ on the same scale? No
- How can we fix this?

Highway Connections

- We can also modify the network to look like this:

$$p = W_2 h$$
$$s = p + b_1$$
$$y = \sigma(s) + \mathbf{W}\mathbf{s}$$
$$e = L2(y, t) = \frac{(t - y)^2}{2}$$

- We could introduce a scaling matrix $\mathbf{W}$
- Even if s is large, the term $\frac{\partial y}{\partial s} = \sigma(s)(1 - \sigma(s)) + \mathbf{W}$ is likely nontrivial
- This strategy is called the Residual Connection, as it increases the chance of gradient flow

Highway Connections

- We can also modify the network to look like this:

$$
\begin{aligned}
p &= W_2 h \\
s &= p + b_1 \\
y &= \mathbf{t(p)}\sigma(s) + \mathbf{(1 - t(p))p} \\
e &= L2(y, t) = \frac{(t - y)^2}{2}
\end{aligned}
$$

- This can be generalized further, as above
- Here, we have a "gate" $\mathbf{t}$ which controls the flow of information further still
- $\mathbf{t}$ can be a sigmoid, for example
- In this case we have $\frac{\partial e}{\partial W_2} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial s}\frac{\partial s}{\partial p}\frac{\partial p}{\partial W_2} + \frac{\partial e}{\partial y}\frac{\partial y}{\partial p}\frac{\partial p}{\partial W_2}$

Highway Connections

- We can also modify the network to look like this:

$$
\begin{aligned}
p &= W_2 h \\
s &= p + b_1 \\
y &= \sigma(p)\sigma(s) + (1 - \sigma(p))p \\
e &= L2(y, t) = \frac{(t - y)^2}{2}
\end{aligned}
$$

- In this case we have $\frac{\partial e}{\partial W_2} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial s}\frac{\partial s}{\partial p}\frac{\partial p}{\partial W_2} + \frac{\partial e}{\partial y}\frac{\partial y}{\partial p}\frac{\partial p}{\partial W_2}$
- The above is $\frac{\partial e}{\partial W_2} = (y - t)\sigma(p)\sigma(s)(1 - \sigma(s)) * 1 * h + (y - t)[\sigma(p)(1 - \sigma(p))\sigma(s) + 1 - \sigma(p) - p\sigma(p)(1 - \sigma(p))]h$

Outline

- Neural Networks
- BackProp - Theory + Examples
- Optimization - Theory + Examples
- Residual Connections
- BatchNorm and LayerNorm
- Regularization - Dropout
- Wave Net

BatchNorm

- We optimize a NN over batches
- At a high level, for each element in a batch we feel it into a nonlinearity, so we get $y = \sigma(x)$
- What if x is large or small?
- As before, we have that if x is large or small, $\sigma(x)$ will be 0 or 1, and this might affect the gradient of parameters in lower layers
- This is since $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$

BatchNorm

- We optimize a NN over batches and batches can change
- A batch might change distribution, so for example x feeding into σ might be all over the place
- The loss will be very bouncy
- Idea: Normalize!

BatchNorm

- For each batch, get μ_B and σ_B^2 which are the mean and variance of the different components of x in the batch (these are vectors, of the same dimension as x)
- Before we feed into σ, normalize the data so that we use $\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
- Now, feed this normalized $\hat{x}$ into σ: less change we'll get extremes!
- But, what if we want to allow the network to learn the scaling and off-centering?
- Introduce two (vector-valued) parameters γ, β
- Use $\gamma\hat{x} + \beta$ as the input to σ
- Maybe $\gamma = 1$ and $\beta = 0$, but let the network learn that

BatchNorm

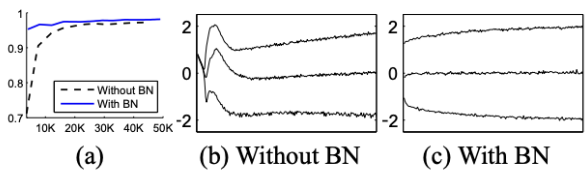
- So the process goes as follows
- Fix a batch $B = \{x_1, \ldots, x_m\}$

$$\mu_B = mean(x_B)$$
$$\sigma_B = stdev(x_B)$$
$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$
$$y = \gamma \hat{x} + \beta$$

- Feed y to a nonlinearity
- Basically, before you run the data through the nonlinearity, apply a batch normalization!

BatchNorm: Benefits

- Higher learning rates!
- Stability!
- When training with Batch Normalization, a training example is seen in conjunction with other examples in the mini-batch, and the training network no longer producing deterministic values for a given training example
- This last point might lead to a bit of regularization



(a) (b) Without BN (c) With BN

Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

BatchNorm: Gradients

- BatchNorm gradients are not that scary: everything is differentiable!
- Here, l is the loss, and we have simplifications like $\frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i}\gamma$

During training we need to backpropagate the gradient of loss ℓ through this transformation, as well as compute the gradients with respect to the parameters of the BN transform. We use chain rule, as follows (before simplification):

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

BatchNorm: Training and Inference

- At training time, we get μ_B and σ_B^2 for each batch and use these statistics
- At inference time, we can recompute these statistics over the entire training set and use these as normalizations
- Or, we can use some sort of running estimate (momentum!)
- For example, if we process mini-batches $B_1, B_2, \ldots, B_T$ in sequence, we can define a mean estimate as $\mu_B = \mu_{B_1}$, then $\mu_B = \alpha\mu_{B_2} + (1 - \alpha)\mu_B$ etc
- We can use μ_B at inference time
- Same for σ_B^2
- **Be careful**: the behavior of this layer depends on what state your model is in!

LayerNorm

- For NLP applications, usually we have batches of sentences and each sentence has different tokens and different lengths
- Each token has an embedding
- We can't really take a mean across all embeddings in the batch, since different embeddings are in different sentences
- We'd lose context if we did this

LayerNorm

- A better way is to normalize **each** token's embedding
- Thus, μ_D and σ_D^2 are the mean and variance across all components of a particular token embedding
- **This method does not depend on the batch size**
- We normalize $\hat{e} = \frac{e-\mu_D}{\sqrt{\sigma_D^2+\epsilon}}$
- Finally, we apply γ and β as before to allow the model to learn the transformation
- We use $\gamma\hat{e} + \beta$ as the final representation of LayerNorm

Outline

- Neural Networks
- Optimization - Theory + Examples
- BackProp - Theory + Examples
- Residual Connections
- BatchNorm and LayerNorm
- Regularization - Dropout
- Wave Net

Drop Out

- The aim of dropout is to make sure the model does not overtrain too much
- At a high level, for each batch we zero out a random proportion (20 %, 30 %) of the neurons
- This effectively zeros out the parameters for these neurons

Drop Out

- You can think of this method kind of like ensembling: on each batch, you are training a slightly different model

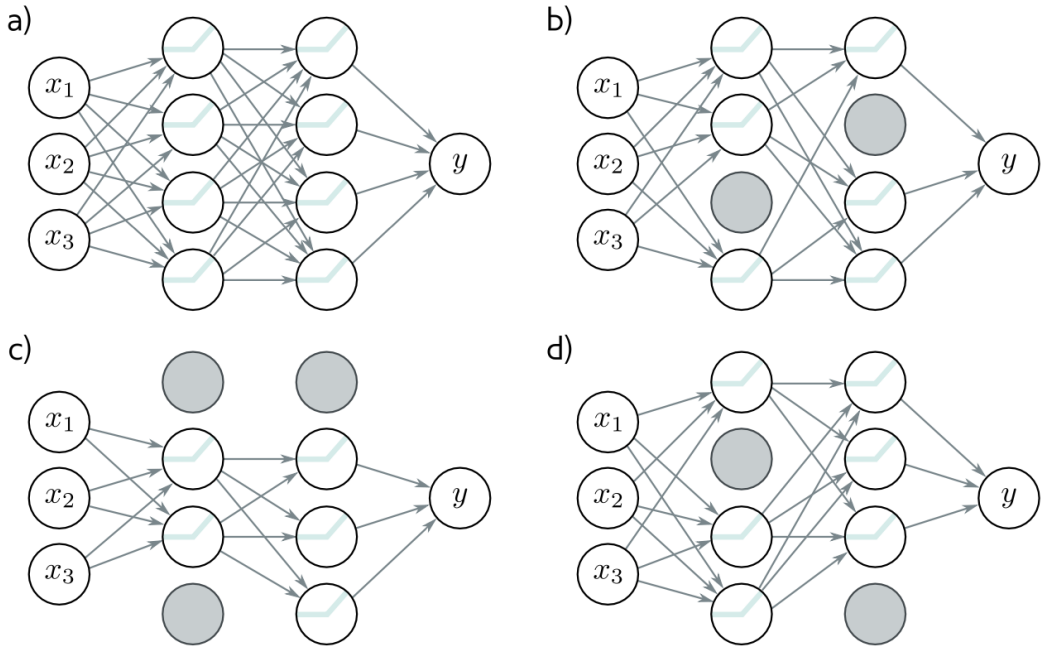


Figure 9.8 Dropout. a) Original network. b-d) At each training iteration, a random subset of hidden units is clamped to zero (gray nodes). The result is that the incoming and outgoing weights from these units have no effect and so we are training with a slightly different network each time.

Drop Out

- You can also think drop out as making the model learn as much as it can in a weakened state: make the best due with what you have, don't get too comfortable using one set of nodes
- This makes the network less dependent on any given hidden unit
- We also encourage the weights to have smaller magnitudes so that the change in the function due to the presence or absence of the hidden unit is reduced

Drop Out

- **Be careful**: the behavior of this layer depends on what state your model is in!

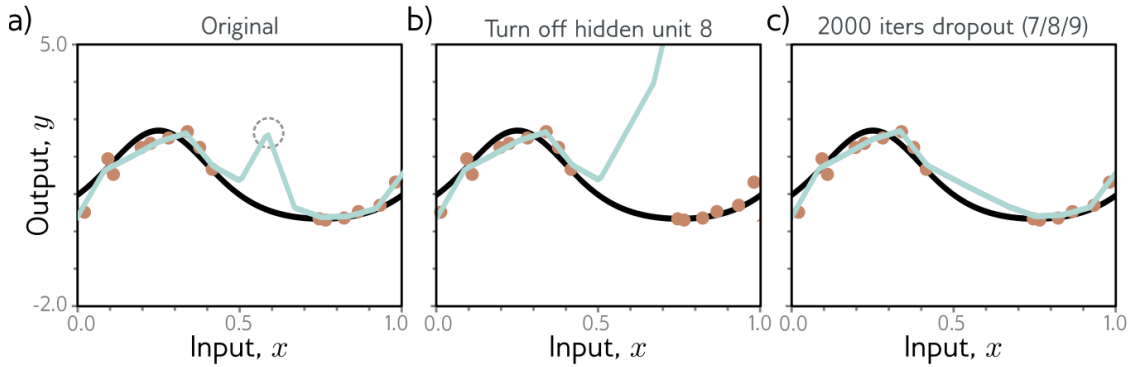


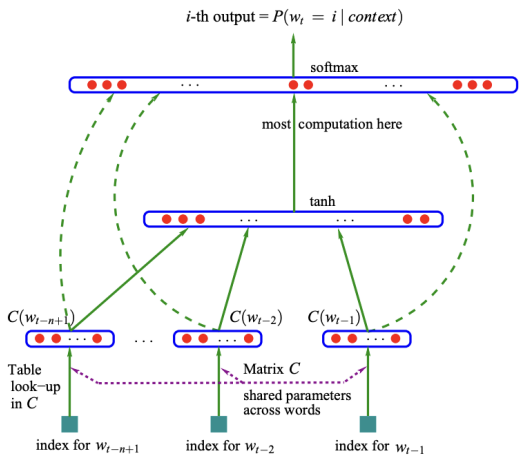
Figure 9.9 Dropout mechanism. a) There is an undesirable kink in the curve caused by a sequential increase in the slope, decrease in the slope (at circled joint), and then another increase to return the curve to its original trajectory. Here we are using full-batch gradient descent and the model already fits the data as well as possible, so further training won't remove the kink. b) Consider what happens if we remove the hidden unit that produced the circled joint in panel (a) as might happen using dropout. Without the decrease in the slope, the right-hand side of the function takes an upwards trajectory and a subsequent gradient descent step will aim to compensate for this change. c) Curve after 2000 iterations of (i) randomly removing one of the three hidden units that cause the kink, and (ii) performing a gradient descent step. The kink does not affect the loss but is nonetheless removed by this approximation of the dropout mechanism.

Outline

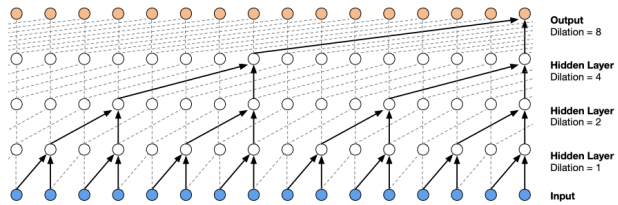
- Neural Networks
- Optimization - Theory + Examples
- BackProp - Theory + Examples
- Residual Connections
- BatchNorm and LayerNorm
- Regularization - Dropout
- Wave Net

Wave Net

- The original language model we looked at tried to predict the next word from a fixed window of K other words
- There was 1 hidden layer
- For HW, we'll adapt the Wave Net idea to built a language model that can have a longer context and multiple layers
- We'll add the tricks discussed and look at their effect



(a) Bengio's Language Model



(b) Wave Net's Audio Model

Figure: Two different Language Model ideas

References

- Neural Machine Translation
- Understanding Deep Learning
- Wave Net
- A Neural Probabilistic Language Model
- Res Net
- Batch Norm
- Layer Norm
- Highway Networks