Kalon Lucas Kelley - 406039396

2 Lecture 2

Supervised learning, we will be given data points $x^{(i)}$ that come with an associated label $y^{(i)}$ which tells us in a classification problem what type of image x is or what value we want to map x to in a regression problem.
Classification: $y^{(i)}$ represent classes or categories that $x^{(i)}$ may belong to. e.g. in CIFAR-10 we are given a bunch of images $\mathbf{x}$ which are $32 \times 32 \times 3$ (tensor) and each image $\mathbf{x}$ will be associated with one label y which there are 10 of. Given an image $\mathbf{x}$ determine which of the 10 classes it belongs to.
Reshape the tensor into a vector $32 \times 32 \times 3 = 3072$ meaning $\mathbf{x} \in \mathbb{R}^{3072}$
Want to define some function $class \leftarrow f(\mathbf{x})$ takes the input image and returns the class the image belongs to.
Because were given pairs of images and labels to train on this is called **supervised**

2.1 Regression

Regression doesn't predict a class but produces a number at the output.
e.g. We want to rent a home in westwood and we are trying to find a relationship between square footage and monthly rent. We want to predict what they rent will be given the square footage and the rent isn't a class it is a real dollar amount hence why this is a **regression** problem and **not classification**.
In machine learning our goal is to learn some function $f(sq\ ft) \rightarrow rent$ that takes the square footage and produces the rent.
We have to define what model we are going to use, the model of f. (e.g. $ax + b$) because if we look at the data graphically it may look like they are pretty well described linearly.
$y = ax + b$ where $\mathbf{y}$ is the monthly rent and $\mathbf{x}$ is the square footage. And both y and x are given to us by data where a and b are knobs we get to tune to get this model to fit as best as possible **hyperparameters**
We could use a different model for f like $b + a_{100}x^{100} + a_{99}x^{99} + ... + a_1 x$ in which case we have many more parameters to tune and can fit the data perfectly but this leads to **overfitting** because the model becomes much worse at generalization.
Later on our f model will be a neural network.
How do we assess how good our model is:
inputs: $x^{(i)}$ (i is for indexing over all inputs)
outputs: $y^{(i)}$
$\hat{y}^{(i)} = b + ax$ we can tell one model is better than another by checking if $\hat{y}^{(i)}$ of a given model is closer than the true outputs $y^{(i)}$
We define some error $\epsilon_i = y^{(i)} - \hat{y}^{(i)}$ if we want to quantify how good a model is we look for the model with the smallest error.
Loss Function: is a function which tells us **quantitatively** how good or how bad our model is $\mathcal{L}$ also sometimes called a cost function.
$\mathcal{L} = \sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})$ the problem with this is that sign is not considered meaning if a data point resides below the prediction and another one resides equally above the prediction their errors will cancel out in the summation making the loss seem like 0. Instead we use the squared error as follows $\mathcal{L} = \sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2$ and oftentimes this is normalized by $\frac{1}{2}$ because we integrate and this allows the square to cancel. It is also normalized by N so this becomes the average loss per example. $\mathcal{L} = \frac{1}{2N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})$. Strictly speaking these constants are not necessary but helpful.
Why not absolute value? you don't get an analytic solution which can be derived by hand but this can still be optimized through gradient descent. Also the squared error magnifies larger distances meaning further away points make loss very bad.
Re-writing more mathematically
$\hat{y} = ax + b$ where a and b are parameters we get to choose to make the model as good as possible
$\hat{y} = \theta^T \hat{\mathbf{x}}$ where $\theta = \begin{bmatrix} a \\ b \end{bmatrix}$ and $\hat{\mathbf{x}} = \begin{bmatrix} x \\ 1 \end{bmatrix}$
The 1 is added to $\hat{\mathbf{x}}$ to ensure b is a bias added at the end
Cost Function $\mathcal{L}(\theta) = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2 = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \theta^T\hat{\mathbf{x}}^{(i)})^2$ leaving out the normalizing factor of N because it doesn't make a functional difference in getting to the optimum.
When $\mathcal{L}$ is larger the model is worse meaning we want to minimize $\mathcal{L}$ which is an optimization problem so we must calculate $\frac{\partial \mathcal{L}}{\partial \theta}$ for our simple example $\mathcal{L}(\theta)$ is a quadratic function of theta (parabola) so there is only one minima which is global. **With neural networks the loss landscapes are much more complicated and countless minima and maxima**

2.2 Vector & Matrix Derivatives

Typically call these derivatives gradients. We have the following syntax y (math notation no bold) is a scalar, $\mathbf{y}$ bold lowercase is a vector, and $\mathbf{Y}$ is a matrix or tensor.
Differentiate a scalar y w.r.t a vector $\mathbf{x}$

$\frac{\partial y}{\partial \mathbf{x}} = \nabla_{\mathbf{x}} y$ e.g. with a scalar y and vector $\mathbf{x} \in \mathbb{R}^n$ then $\nabla_{\mathbf{x}} y = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix}$ we know that the derivative of a scalar with respect to a scalar tells us ho much moving the second one will move the first. (e.g. Δy due to Δx_1 is $\approx \frac{\partial y}{\partial x_1}\Delta x_1$ if $\frac{\partial y}{\partial x_1} = 0.5$ and we change (wiggle) x_1 by 0.01 then we would expect $\Delta y \approx 0.005$
What we can now do $\Delta y \approx \sum_{i=1}^{n} \frac{\partial y}{\partial x_i}\Delta x_i$ which can be written as a **dot product** telling us concisely how y will change when we wiggle a vector $\mathbf{x}$
$\Delta y \approx (\nabla_{\mathbf{x}} y)^T \Delta \mathbf{x}$ which generalizes derivative concepts from calculus to vectors. Think of this as how changing our **parameters** changes our **loss**
Example:

If $y = \theta^T \mathbf{x} = \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n$ then $\nabla_{\mathbf{x}} y = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} = \theta$

Another example: if $f(x) = \mathbf{x}^T \mathbf{A} \mathbf{x}$ what is $\nabla_{\mathbf{x}} f(x)$

$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & ... & a_{1,n} \\ a_{2,1} & a_{2,2} & ... & a_{2,n} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & ... & a_{n,n} \end{bmatrix}$ We can re-write the product in summation form as
$\sum_{i=1}^{n}\sum_{j=1}^{n} x_i \times a_{i,j} \times x_j$ and now differentiate each term as follows
$\frac{\partial f(x)}{\partial x_1} = 2a_{1,1}x_1 + \sum_{j=2}^{n} a_{1,j}x_j + \sum_{i=2}^{n} a_{i,1}x_1$ because those are all the terms which can contain an x_1 this can be repeated for all x_i and this can even be simplified further by putting the first term into the second two as follows. $\frac{\partial f(x)}{\partial x_1} = \sum_{j=1}^{n} a_{1,j}x_j + \sum_{i=1}^{n} a_{i,1}x_1$ which can be written in matrix sum notation as $(\mathbf{A}\mathbf{x})_1 + (\mathbf{A}^T\mathbf{x})_1$ and we can put all of these elements for all x_i into a vector for the final gradient $\nabla_{\mathbf{x}} f(x) = (\mathbf{A} + \mathbf{A}^T)\mathbf{x}$ and in the case where $\mathbf{A}$ is symetric this is equivalent to $2\mathbf{A}\mathbf{x}$.
This can be sanity checked by assuming $n = 1$ in which case all our terms are scalars and we know that $\frac{\partial ax^2}{\partial x} = 2ax$
Matrix Derivatives: If $\mathbf{A} \in \mathbb{R}^{m \times n}$ then $\nabla_{\mathbf{A}} y \in \mathbb{R}^{m \times n}$ where each element is the derivative of y w.r.t the same element of the matrix.
We can no go back and rewrite the loss function (cost function) as follows
$\mathcal{L}(\theta) = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \theta^T\hat{\mathbf{x}}^{(i)})^2 = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \theta^T\hat{\mathbf{x}}^{(i)})^T(y^{(i)} - \theta^T\hat{\mathbf{x}}^{(i)})$
Where we can reorder the $\theta^T\hat{\mathbf{x}}^{(i)}$ by applying a transpose becoming $(\hat{\mathbf{x}}^{(i)})^T\theta$ then vectorize it to remove the summation.

2.3 Denominator vs Numerator layout

Denominator layout: $\mathbf{A} \in \mathbb{R}^{m \times n}$ then $\nabla_{\mathbf{A}} y \in \mathbb{R}^{m \times n}$ and $\mathbf{x} \in \mathbb{R}^{n \times 1}$ then $\nabla_{\mathbf{x}} y \in \mathbb{R}^{n \times 1}$
Numerator layout: $\mathbf{A} \in \mathbb{R}^{m \times n}$ then $\nabla_{\mathbf{A}} y \in \mathbb{R}^{n \times m}$ and $\mathbf{x} \in \mathbb{R}^{n \times 1}$ then $\nabla_{\mathbf{x}} y \in \mathbb{R}^{1 \times n}$
They are the exact same values between numerator and denominator but just transposed.

3 Lecture 3

Our re-written loss function from lecture 2 vectorized $\mathcal{L} = \frac{1}{2}(Y - \mathbf{X}\theta)^T(Y - \mathbf{X}\theta)$ where $Y \in \mathbb{R}^N$ and is the vector of all $y^{(i)}$, $\mathbf{X} \in \mathbb{R}^{N \times 2}$ and is the vector of vectors $\hat{\mathbf{x}}^{(i)T}$
$\mathcal{L}(\theta) = \frac{1}{2}(Y^TY - Y^T\mathbf{X}\theta - \theta^T\mathbf{X}^TY + \theta^T\mathbf{X}^T\mathbf{X}\theta) = \frac{1}{2}(Y^TY - 2Y^T\mathbf{X}\theta + \theta^T\mathbf{X}^T\mathbf{X}\theta)$
Recall $\frac{\partial \mathbf{z}^T\theta}{\partial \theta} = \mathbf{z}$ and we'll say $Y^T\mathbf{X} = \mathbf{z}^T$ and using $\nabla_\theta(\theta^T\mathbf{A}\theta) = (\mathbf{A} + \mathbf{A}^T)\theta$ where $\mathbf{A} = \mathbf{X}^T\mathbf{X}$
$\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = \frac{1}{2}(0 - 2\mathbf{X}^TY + (\mathbf{X}^T\mathbf{X} + \mathbf{X}^T\mathbf{X})\theta) = -\mathbf{X}^TY + \mathbf{X}^T\mathbf{X}\theta$ then setting that to 0 and solving we get $\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^TY$ and this is **Least Squares**
$\mathbf{X}^T\mathbf{X} \in \mathbb{R}^{2 \times 2}$ if we have more features than samples then $\mathbf{X}^T\mathbf{X}$ will not be invertable so we cannot use the same formula for θ

3.1 Least Squares for Higher Order Polynomials

We can use the same math as before for higher order polynomials with $\hat{\mathbf{x}} = \begin{bmatrix} x^n \\ x^{n-1} \\ \vdots \\ x \\ 1 \end{bmatrix}$ and $\theta = \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_1 \\ b \end{bmatrix}$

where our polynomial is $y = b + a_1x + a_2x^2 + ... + a_nx^n$ leaving our same equation $\theta^T\hat{\mathbf{x}}$

A higher degree polynomial will *always* fit the provided data as well as a lower order polynomial or better

3.2 Overfitting

If we make the order of the polynomial higher we will fit the data better but may not be as good at generalizing, we don't care about how well it fits our training data but how well it is able to generalize on new unseen data.

- **Training Data:** data that is used to learn the parameters of your model
- **Testing Data:** data that is excluded in training and used to score your model (unseen)

Overfitting occurs when a model has very low training error but high testing error (does not generalize) lots of data helps to mitigate overfitting even with higher order polynomials. (When lots of data is available we will be able to use more complex models)

3.3 Underfitting

Making a model *overly simple* will underfit the data meaning the model has both a high training and testing error not generalizing or fitting the training data well at all. This is because the model is not expressive enough.

3.4 Hyperparameters

Things we choose beforehand that have to do with the model itself (like the order of the polynomial) and these hyperparameters can have a profound affect on the model fitting.

3.5 Data Sets

There are 3 types of data sets or partitions which we need for creating a model.

- **Training Data:** Data that is ued to learn the parameters of your model
- **Validation Data:** Data that is used to optimize the hyperparameters of your model, avoiding the potential of overfitting to a nuanced dataset
- **Testing Data:** Data that is used to score the final model after all optimizations

3.6 K-fold Cross Validation

Assuming we have a **separate** testing set, to train the model a common approach is k-fold cross validation which is used as follows.

- Let the training dataset contain N examples (e.g. 800)
- Split the data into k equal sets containing $\frac{N}{k}$ examples each each of which is called a fold
- $k-1$ folds are datasets used for training the model parameters (e.g. 3 folds 600 examples) for θ
- The remaining fold is a **validation** dataset used to evaluate the model
- Repeatedly train the model by choosing which folds comprise the training folds and testing fold

Why we don't optimize hyperparameters for testing data: If we did then the model may be bad at generalizing for future unseen data (again overfitting)

We have discussed wanting to *minimize* a mean-square error or distance metric, but another metric we may want to *maximize* is the probability of having *observed* the data. In this framework the data is modeled to have some distribution with parameters. We choose the parameters to maximize the probability og having observerd our training data. (**maximum likelihood**)

Example of this is **softmax classifier** which is a linear classifier

3.7 Image Classification

Images are seen by our computers as a 3D tensor ($w \times h \times 3[rgb]$)

Since the image is stored as an array of numbers there are several challenges associated with image classification

- **Viewpoint Variation:** viewing an object from different angles and taking the picture can result in the pixels from the same location between two pictures of the object having drastically different values
- **Illumination:** If there is a lot of light the pixels may be closer to 255 than if there isn't
- **Deformation:** We may classify a cat as having 4 legs and pointy ears but it isn't always true
- **Occlusion:** A cat may be hiding behind something so all of its features are not visible
- **Background Clutter:** Blending in with the backgroud of the image
- **Interclass Variation:** Cats have different shapes sizes patterns etc (hard to classify)

When trying to classify an image there are several approaches but we will be using the **data driven** approach where we let a machine learning algorithm see a lot of the data and learn a function mapping the image to a class.

Train: get data x, y learn model parameters from the data θ. In deep neural networks these θs result in learning features that are optimized to classify an image).

Features are $\hat{\mathbf{x}}$ in deep NN's these features are inferred from the data.

Test: copy the model from training and we use it to predict the class of a new image.

3.8 K nearest neighbors

Given a training set of input vectors $\{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$ and their corresponding classes $\{y^{(1)}, y^{(2)}, ..., y^{(m)}\}$, in the example of CIFAR-10 $x \in \mathbb{R}^{3072}$ and $y^{(i)} \in \{1, 2, ..., 10\}$. If we have a datapoint x^{new} we want to classify it.

Intuitavely, k-nearest neighbors says to find the k closest points (nearest neighbors) in the training set, according to an appropriate metric. Each of its k nearest neighbors then vote according to what class it is in, and x^{new} us assigned to be the class with the most votes.

- Choose an appropriate distance metric, $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ returning the distance between $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$. e.g. $||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||_2 = \sqrt{\sum_{k=1}^{n}(\mathbf{x}_k^{(i)} - \mathbf{x}_k^{(j)})^2}$. We could also use the 1-norm which is absolute distance, or the ∞-norm (max distance between entries in the vectors). **hyperparameter**
- Choose the number of nearest neighbors, k **hyperparameter**
- Take desired test data point, $\mathbf{x}^{new}$ and calculate $d(\mathbf{x}^{new}, x^{(i)})$ for $i = 1, ..., m$
- With $\{c_1, ..., c_k\}$ denoting the k indices corresponding to the k smallest distances classify $\mathbf{x}^{new}$ as the class that occurs most frequently in $\{y^{c_1}, ..., y^{c_k}\}$

How do we train the classifier?

Just cache the training data and labels.

+ Simple, Fast, very easy
− Memory Intensive (we need to store all input data)

How do we test a new data point?

Compute the distances from the test data-point to every other data-point in the training set, sort the distances and take the k nearest distances to classify the test data-point.

+ Simple
− Slow (scale with the amount of training data)

In general we are ok with a long training cost as long as the test inference time is quick because the test time is what is happening in deployment.

4 Lecture 4

Why might k-nearest neighbors not be sufficient for image classification?

It computes a difference in the values of the pixels but these differences are not the same as semantic differences.

Curse of dimensionality:

- Images are very high-dimensional vectors (e.g. CIFAR-10 is 3072 dimensional)
- Notions of *distance* become less intuitive in higher dimensions
- Distances in some dimensions matter more than others (but in k-nn there is no preference)
- In higher-dimensional space, the volume increases exponentially, this leaves a lot of empty space so nearest neighbors may not be so close.

4.1 Classifiers based on Linear Classification

Perhaps a better way would be to develop a *score* for an image coming from each class, and then pick the class that achieves the highest score. **Neural networks** are built on linear classifiers with each layer being composed of one followed by a nonlinear function.

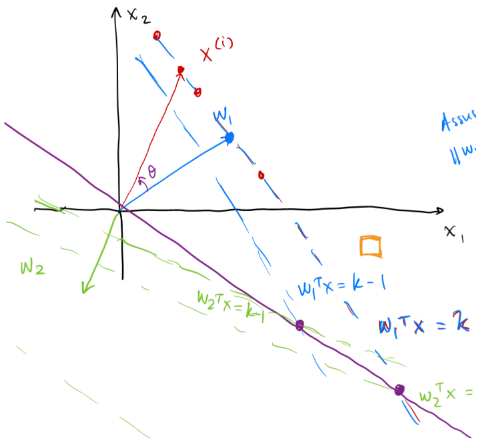
Example: Consider a matrix $\mathbf{W} = \begin{bmatrix} - & \mathbf{w}_1^T & - \\ & \vdots & \\ - & \mathbf{w}_c^T & - \end{bmatrix}$ where $c = 10$ (number of classes). Then $\mathbf{W} \in \mathbb{R}^{c \times N}$.

Let $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ where $\mathbf{b}$ is a vector of bias terms, then $\mathbf{y} \in \mathbb{R}^c$ is a vector of scores with its ith element corresponding to the score of $\mathbf{x}$ being in class i. The chosen class corresponds to the index of the highest score in $\mathbf{y}$. We get the dimensionality $\mathbf{W}(10 \times 3072)\mathbf{x}(3072 \times 1) + \mathbf{b}(10 \times 1) = \mathbf{y}(10 \times 1)$

Where $\mathbf{y} = \begin{bmatrix} \mathbf{w}_1^T\mathbf{x} + b_2 \\ ... \end{bmatrix}$ where y_i is the score of the image x belonging to the ith class. So if y_2 had the highest score then the image would be classified as belonging to class 2 (in CIFAR-10 automobiles).

Since the score of a given class is determined by the dot product $\mathbf{w}_i^T\mathbf{x}$ then we know the score is high when the two look similar, meaning if we visualize the weights of a given class $\mathbf{w}_i$ we can see what the *average* image of that class looks like.

2d example: $\mathbf{x} \in \mathbb{R}^2$ so $y_1 = \mathbf{w}_1^T\mathbf{x} = \|\mathbf{w}_1\|\|\mathbf{x}\|cos(\theta)$ assuming $\|\mathbf{w}_1\| = 1$ then we can simplify $= \|\mathbf{x}\|cos(\theta)$



Changing the assumption of $\|\mathbf{w}_1\|$ to be something larger like 2 just changes the decision boundary

Where might linear classifiers fail? When the data are not linearly separable, imagine and XOR. Another example is when data is radially oriented but in this case we can transform the representation into polar coordinates making the circles vertical lines.

Deep learning layers are computing non-linear transformations of the data so that at the end they become linearly separable by the softmax classifier.

Now how do we take the scores we received (analog in value) and turn the into an appropriate **loss function** for us to optimize so we can learn $\mathbf{W}$ and $\mathbf{b}$ properly.

We do this through maximum likelihood optimization (maximize the probability of having observed the data).

Example: We get data for a coin-flip and we want to choose the probability of heads that maximizes the likelihood of having observed our data. Data $= \{H, T, H, H, T, T, H, T\}$

Model $x^{(i)} = \begin{cases} H(1) & w.p\ \theta \\ T(0) & w.p\ 1-\theta \end{cases}$ so we want a θ with that results in the highest likelihood.

- **Model 1:** $\theta = 1$ we get $1 * 0 * 1 * 1 * 0 * 0 * 1 * 0 = 0$
- **Model 2:** $\theta = 0.75$ we get $(0.75)^4 * (0.25)^4 = 0.00124$
- **Model 3:** $\theta = 0.5$ we get $(0.5)^4 * (0.5)^4 = 0.0039$ which is the highest likelihood (**maximum**)

We can define a likelihood $\mathcal{L} = \theta^\theta(1-\theta)^4$ and then we can differentiate the *log* (**because log is monotonically increasing**) of the likelihood with respect to theta and set to 0 to find θ maximizing the likelihood of having observed the given data.

In the case of the **softmax** classifier there is not one global optima (maxima in this case) so we cannot just take the derivative but we use gradient descent instead.

4.2 Chain rule for Probability

Notations for the class. $Pr(A = a) = p_A(a) = p(a)$ and $Pr(A = a, B = b) = p_{A,B}(a, b) = p(a, b)$

Chain Rule Example $Pr(A = a, B = b) = Pr(A = a)Pr(B = b\ given\ A = a)$ can be written as $p(a, b) = p(a)p(b|a) = p(b)p(a|b)$

We can decompose the probability $p(a, b, c)$ into $p(c) * p(a|c) * p(b|a, c)$ **or** $p(a, c) * p(b|a, c)$ meaning every single outcome goes once in front of the condition bar and every subsequent time they go behind the condition bar.

example

$p(b, c|d, e) = \frac{?}{p(d)*p(e|d)}$ if we simplify the bottom into $p(d, e)$ and then move it to the other side and simplify we get $? = p(b, c, d, e)$

4.3 Softmax Function

We normalize the scores by turning them into probabilities. The softmax function transforms the class score into a probability via $\text{softmax}_i(\mathbf{x}) = \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})}}$ for $a_i(\mathbf{x}) = \mathbf{w}_i^T\mathbf{x} + w_i$ and c being the number of classes, and $a_i(\mathbf{x}) = \mathbf{y}_i = \mathbf{w}_i^T\mathbf{x} + b_i$ meaing $\mathbf{a}_i$ is the score if the ith class for the image $\mathbf{x}$ we can also write it as $\mathbf{a}_i(\mathbf{x}) = \tilde{\mathbf{w}}_i^T\mathbf{x}$ we just removed b_i by concatenating it at the end of each vector in $\mathbf{W}$ and adding a 1 to the end of the matrix $\mathbf{x}$.

We can see that $0 \leq \text{softmax}_i(\mathbf{x}) \leq 1$ and that the sum of all softmax values is 1.

We use e specifically for connections to information theory when calculating things like the loss which is cross entropy loss, and also since we will be taking the log which will be easy with the e.

$\text{softmax}_i(\mathbf{x}^{(j)}) = Pr(y^{(j)} = i|\mathbf{x}^{(j)}, \theta)$ meaning the softmax of an image, is the probability that the image belongs to the class i

4.4 Softmax Classifier

Although we know the softmax function, how do we specify the *obejective* or **loss function** to be optimized with respect to θ?

Maybe we should choose the parameters θ that maximize the likelihood of having seen the data. Assuming the samples, $(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(m)}, y^{(m)})$ are iid, this corresponds to maximizing: $p(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}, y^{(1)}, ..., y^{(m)}|\theta) = \prod_{i=1}^{m} p(\mathbf{x}^{(i)}, y^{(i)}|\theta) = \prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta)p(y^{(i)}|\mathbf{x}^{(i)}, \theta)$ where $y^{(i)}$ is the class of the image $\mathbf{x}^{(i)}$ and $\theta = \{\mathbf{W}, \mathbf{b}\}$.

We decompose it in this way because we defined $\text{softmax}_i(\mathbf{x}^{(j)}) = p(y^{(j)}|\mathbf{x}^{(j)}, \theta)$.

The assumption that they are iid is necessary for using the product notation (this assumption is broken in video) or else we would need the first image and class to be behind the condition bar of every subsequent probability.

We want to choose a θ that makes the data as likely as possible. $\underset{\theta}{\arg\max} \prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta)p(y^{(i)}|\mathbf{x}^{(i)}, \theta)$ which is equivalent to $\underset{\theta}{\arg\max} \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}, \theta)$ because the probability $p(\mathbf{x}^{(i)}|\theta)$ has no dependence on θ.

Since *log* is monotonic we can apply it to the probabilities and this turns our product into a sum, so we get $\underset{\theta}{\arg\max} \sum_{i=1}^{m} log(\text{softmax}_{y^{(i)}}(\mathbf{x}^{(i)}))$ with the notation $\text{softmax}_{y^{(i)}}(\mathbf{x}^{(i)})$ we are saying for the ith example we have an image and a label $y^{(i)}$ and for that image evaluate the probability that the image belongs to that class.

$\underset{\theta}{\arg\max} \sum_{i=1}^{m}[a_{y^{(i)}}(\mathbf{x}^{(i)}) - log(\sum_{j=1}^{c} e^{a_j(x^{(i)})})]$ which we can re-write and add a normalization factor $\frac{1}{m}$ as $\underset{\theta}{\arg\min} \frac{1}{m} \sum_{i=1}^{m}[log(\sum_{j=1}^{c} e^{a_j(x^{(i)})}) - a_{y^{(i)}}(\mathbf{x}^{(i)})]$ because $\underset{\theta}{\arg\max} f(\theta) = \underset{\theta}{\arg\min} - f(\theta)$

This is loss if called the **cross entropy loss**.

4.5 Softmax Classifier: Intuition

When maximizing the softmax the term $a_{y^{(i)}}(\mathbf{x})$ is made larger and the term $log(\sum_{j=1}^{c} e^{a_j(x^{(i)})})$ is made smaller. The latter term can be approximated by $\max_j a_j(\mathbf{x})$.

- If $a_{y^{(i)}}(\mathbf{x})$ produces the largest score then the log likelihood is approximately 0
- If $a_j(\mathbf{x})$ produces the largest score for $j \neq i$, then $a_{y^{(i)}}(\mathbf{x}) - a_j(\mathbf{x})$ is negative making the log likelihood negative

So this helps the intuition that when the correct class has the largest score the loss is minimized.

Overflow of softmax Since a score can be very large causing the value in the numerator of softmax to be even larger it may overflow storage so it is standard practice to normalize by subtracting off the largest score because this has no effect on the resulting softmax.

5 Lecture 5

5.1 Softmax Continued

Our goal is to optimize and objective function **loss function** $f(x)$ ($\mathcal{L}(\theta)$ (without loss of generality)

- From basic calculus we know that the derivative of a function tells us the slope of the function at the point.
- For a small enough ϵ, $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$ telling us how to reduce (or increase) $f(\cdot)$ for small enough steps. This means that we are assuming for a small enough step along the linear approximation we expect the function to be about that value.
- Recall that when $f'(x) = 0$ we are at a critical point can be local/global minimum/maximum or a saddle point.

Terminology

- **Global Minimum:** The point, x_g, that achieves the absolute lowest value of $f(x)$
- **Local Minimum:** The point x_l, that is a critical point of $f(x)$ and is lower than neighboring points
- **Saddle Point:** Critical point of $f(x)$ that are not local minima or maxima, neighboring points are both greater than and less than $f(x)$

Since these loss landscapes are so complicated in general we will be finding local minima but this isn't actually as bad as you may think.

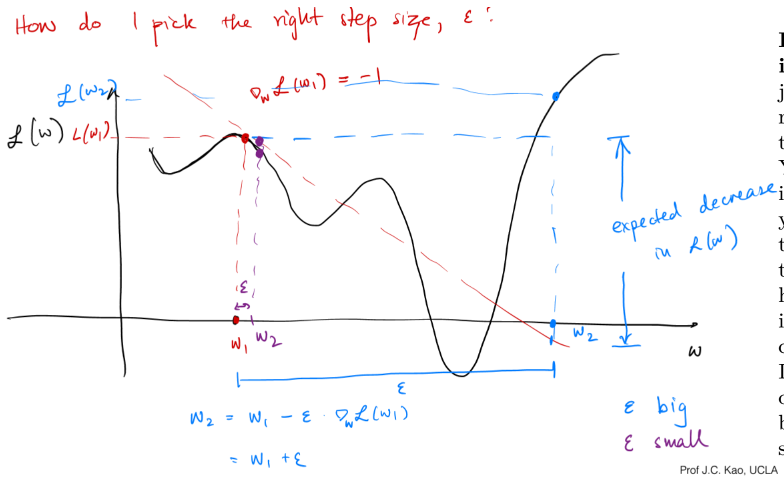
The Intuition is that the loss function is a function of our parameters and with lots of parameters > 100000 the likelihood of being in a local minima (where you cannot search any of the dimensions and get a lower loss) and there being a lower local minima is statistically improbable.

5.2 Gradient Descent

Recall the gradient $\Delta_{\mathbf{x}} f(\mathbf{x})$ is a vector whose ith element is the partial derivative of $f(\mathbf{x})$ w.r.t x_i the ith element of $\mathbf{x}$. Concretely, for $\mathbf{x} \in \mathbb{R}^n$

- The gradient tells us how a small change in $\Delta\mathbf{x}$ affects $f(x)$ through

  $f(\mathbf{x} + \Delta\mathbf{x}) \approx f(\mathbf{x}) + \Delta\mathbf{x}^T \nabla_{\mathbf{x}} f(\mathbf{x})$
- The directional derivative of $f(x)$ in the direction of the unit vector $\mathbf{u}$ is given by $\mathbf{u}^T \nabla_{\mathbf{x}} f(\mathbf{x})$
- The directional derivative tells us the slope of f in the direction of $\mathbf{u}$ we want a $\mathbf{u}$ minimizing $f(\mathbf{x})$.
- To minimize $f(\mathbf{x})$ we want to find the direction in which $f(\mathbf{x})$ decreases the fastest. We do this by finding the direction $\mathbf{u}$ which minimizes the directional derivative. $\underset{\mathbf{u}, \|\mathbf{u}\|=1}{\min} \mathbf{u}^T \nabla_{\mathbf{x}} f(\mathbf{x}) = \underset{\mathbf{u}}{\min} \|\nabla_{\mathbf{x}} f(\mathbf{x})\| cos(\theta)$ where θ is the angle between the vectors $\mathbf{u}$ and $\nabla_{\mathbf{x}} f(\mathbf{x})$
- The quantity is minimized for $\mathbf{u}$ pointing in the opposite direction of the gradient, so that $cos(\theta) = -1$
- Hence we arrive at gradient descent. To update $\mathbf{x}$ so as to minimize $f(\mathbf{x})$ we repeatedly calculate $\mathbf{x} := \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$ where ϵ is typically called the *learning rate*. It can change over iterations and setting it appropriately is an important part of deep learning.



Interpreting Learning Rates: Why not just have small learning rate? it takes too long to get to minima.
You can get a good indication by checking your loss with iterations. If it explodes then learning rate is too high, if it goes slowly it is too low, if it stays constant it is too high. It is perfect when it decreases fast at the beginning and slowly slows down

Why don't we use numerical gradient? (limit thing)

- We would have to calculate the gradient for every parameter (bad with lots of parameters)
- Evaluating f may be expensive (lots of matrix multiplications)

In optimization we differentiate the cost function f w.r.t parameters and the resulting gradient is a function of the training data. So we can think of each data point as providing a noisy estimate of the gradient at that point.

5.3 Batch vs Minibatch

It is expensive to calculate the gradient exactly because it requires evaluating the model on all m examples in the data set

- **Batch Algorithm:** Use all m (e.g. 50000) examples in the training set to calculate a gradient
- **Minibatch Algorithm:** Approximates the gradient by calculating it using k training examples (usually called **stochastic gradient descent**). We need the class statistics to be as close as the statistics of the full batch as possible because otherwise the model might be trained better for classifying one specific class.
- **Stochastic Algorithm:** Approximates the gradient by calculating it over some example

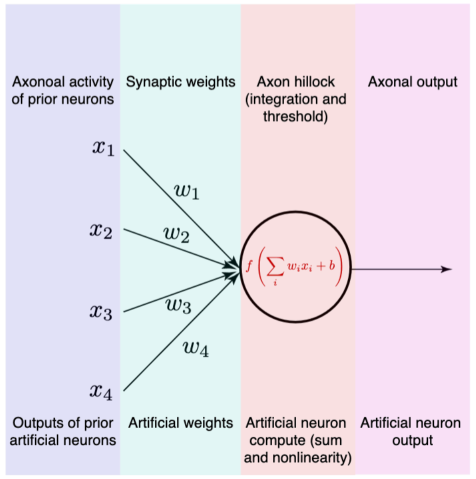
To get a more robust estimate of the gradient we need to use lots of data samples.
$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} log[p_{model}(\mathbf{x}^{(i)}, y^{(i)}|\theta)$ which is the same as the expected value of the log likelihood. Then taking its gradient we get $\nabla_\theta J(\theta) = \nabla_\theta \frac{1}{m}\sum_{i=1}^{m} log[p_{model}(\mathbf{x}^{(i)}, y^{(i)}|\theta) \approx \mathbb{E}[\nabla_\theta log(p_{model}(\mathbf{x}^{(i)}, y^{(i)}|\theta))]$ where m is the batch size.
As it turns out as well sometimes minibatch is a better approach than the full batch algorithm because having noise leads to more robust answers (generalization benefit).

5.4 Neural Networks

Inspired from neurons in neuroscience, neurons are the main signaling units of the nervous system.
Neurons have four regions:

1. Cell body (soma) - metabolic center, with nucleus etc
2. Dendrites - tree like structure for receiving **input** signals
3. Axon Hillock - integrative part of the neuron(sums up all inputs from dendrites) and checks if it is bigger than a threshold if so it fires a spike
4. Axon - single, long, tubular structure for sending **output** signals
5. Presynaptic terminals - sites of communication to next neurons

The spikes are probabilistic (different every trial even for same stimulus) we think of the spikes as reflecting an underlying rate. This rate is what the neural-networks are *encoding*


Axonal activity of prior neurons
Synaptic weights
Axon hillock (integration and threshold)
Axonal output
x_1
x_2
x_3
x_4
w_1
w_2
w_3
w_4
$f\left(\sum_i w_i x_i + b\right)$
Outputs of prior artificial neurons
Artificial weights
Artificial neuron compute (sum and nonlinearity)
Artificial neuron output

The f can be the threshold function from neurons but a better activation function is often chosen (can be non-linear)
Nomenclature

- The first layer of a NN is the input layer (represented by x)
- The last layer is the output layer (represented by z) think of these as the scores that go to a softmax classifier (often called logits)
- Intermediate layers are called hidden layers represented with variable $\mathbf{h}$. For a hidden layer $\mathbf{h}$ the values are calculated as follows $f(\mathbf{Wx} + \mathbf{b})$ where f is the activation function.
- When we specify that a network has N layers, this does not include the input layer

Intuition behind going to lower dimensional output layer (4 hidden neurons to 2 output neurons). The number of outputs is the number of classes we have (10 in CIFAR-10) we may want a higher dimensional feature space because it makes separating these 10 classes easier.
Fully Connected Neural Networks (sometimes known as Multilayer Perceptron [MLP]) have a connection from every neuron in the previous layer to all neurons in the next layer.
To calculate the number of learnable parameters we need the number of weights, $\#\mathbf{x} \times \#\mathbf{h}_1 + \#\mathbf{h}_1 \times \#\mathbf{h}_2 + ... + \#\mathbf{h}_{N-1} \times \#\mathbf{z}$, plus the number of biases $\#\mathbf{h}_1 + ... + \#\mathbf{h}_{N-1} + \#\mathbf{z}$
First layer: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$ **Second layer:** $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$ **(output) layer:** $\mathbf{z} = \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3$

5.5 Activation Function

What if $f()$ is linear? (e.g. $f(x) = x$. Layer 1: $\mathbf{h}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$, Layer 2: $\mathbf{h}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2$ ... Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_n$.
Any composition of linear functions can be reduced to a single linear function $\mathbf{z} = \mathbf{Wx} + \mathbf{b}$ where $\mathbf{W} = \mathbf{W}_N...\mathbf{W}_2\mathbf{W}_1$ and $\mathbf{b} = \mathbf{b}_N + \mathbf{W}_N\mathbf{b}_{N-1} + ... + \mathbf{W}_N...\mathbf{W}_3\mathbf{b}_2 + \mathbf{W}_N...\mathbf{W}_2\mathbf{b}_1$
If f is identity or linear our output $\mathbf{z}$ is still an affine function of the input $\mathbf{x}$ and after passing $\mathbf{z}$ into the softmax we are still just doing linear classification. (nothing more than a linear classifier)
This may be useful in some contexts dim($\mathbf{h}$) << dim($\mathbf{x}$) (dimensionality reduction), this corresponds to finding a low-rank representation of the inputs.
Autoencoder want the loss to be $\mathcal{L} = \|\mathbf{z} - \mathbf{x}\|^2$ (we want $\mathbf{z}$ to look like $\mathbf{x}$)
Introducing Nonlinearity to increase network capacity by adding nonlinear $f(\cdot)$ at the output of each artificial neuron.
Layer 1: $\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$, Layer 2: $\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$ ... Layer N: $\mathbf{z} = \mathbf{W}_N\mathbf{h}_{N-1} + \mathbf{b}_n$.
$f(\cdot)$ is typically called an *activation function* and is applied element-wise to the input.

6 Lecture 6

6.1 What nonlinearity/activation function

One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm.
Sigmoid Unit:
$\sigma(x) = \frac{1}{1+exp(-x)}$ where $\frac{\partial\sigma(x)}{\partial(x)} = \sigma(x)(1 - \sigma(x))$

Our cost/loss function $\mathcal{L}(\mathbf{w})$ is a function of our weights $\mathbf{w}$ in this example we'll have $\mathbf{w}$ as a vector not matrix $\sigma(\mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{x}+\mathbf{b})$ and when we are doing gradient descent we update our weights $\mathbf{w}$ as follows $\mathbf{w} \leftarrow \mathbf{w} - \epsilon\frac{\partial\mathcal{L}}{\partial\mathbf{w}}$ and from the quote above we want $\frac{\partial\mathcal{L}}{\partial\mathbf{w}}$ to be large.
By the chain rule $\frac{\partial\mathcal{L}}{\partial\mathbf{w}} = \frac{\partial\sigma(\mathbf{w})}{\partial\mathbf{w}}\frac{\partial\mathcal{L}}{\partial\sigma(\mathbf{w})}$
The input to our sigmoid is $\mathbf{w}^T\mathbf{x}+b_1$ there are locations in the sigmoid where the input (if it's too big or too small) then our $\frac{\partial\sigma(\mathbf{w})}{\partial\mathbf{w}}$ will be very small making the overall gradient very small meaning there will be little to no learning.

+ around $x = 0$ the unit behaves linearly, and it is differentiable everywhere
− At extremes the unit *saturates* and has zero gradient resulting in no learning
− The sigmoid unit is non-negative. SGD with the sigmoid unit will zig-zag (in practice not too much concern) e.g. if $f(\mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{h}_1 + b)$ and $z = \mathbf{w}^T\mathbf{h}_1 + b$ we can say $\frac{\partial f(\mathbf{w})}{\partial\mathbf{w}} = \frac{\partial z}{\partial\mathbf{w}}\frac{\partial f(\mathbf{w})}{\partial z} = \sigma(z)(1-\sigma(z))\mathbf{h}_1$ where the entire gradient $\frac{\partial f(\mathbf{w})}{\partial\mathbf{w}}$ is ≥ 0. Remember we calculate the gradient descent steps as $\frac{\partial\mathcal{L}}{\partial\mathbf{w}} = \frac{\partial f(\mathbf{w})}{\partial\mathbf{w}}\frac{\partial\mathcal{L}}{\partial f(\mathbf{w})}$ where the second term is a scalar with values ≥ 0 or ≤ 0 meaning the steps made by gradient descent can either be all weights in the negative direction or all weights in the positive direction. So if the minima is in a place where some weights are negative and some are positive it will have to zig-zag.

Hyperbolic Tangent, $\tanh(x) = 2\sigma(2x) - 1$, the hyperbolic tangent is a zero-centered sigmoid-looking activation. And $\frac{\partial\tanh(x)}{\partial x} = 1 - \tanh^2(x)$

+ Around $x = 0$ unit behaves linearly, differentiable everywhere, and zero-centered (no zig-zag)
− Like the sigmoid unit, when a unit saturates (values grow larger or smaller), no additional learning occurs (gradients 0)

Rectified Linear Unit, $\text{ReLU}(x) = max(0, x)$ and $\frac{\partial\text{ReLU}(x)}{\partial x} = \begin{cases}1 & x > 0\\ 0 & x < 0\end{cases}$ this function is not differentiable at $x = 0$ However we can define its subgradient to be between [0,1] at $x = 0$.
When x is positive the gradient is 1

+ In practice, learning with ReLU converges faster than sigmoid and tanh (6x faster)
+ When a unit is active, it behaves as a linear unit
+ The derivative at all points (besides 0) is 0 or 1, no saturation if $x > 0$
− $\text{ReLU}(x)$ is like sigmoid it zig-zags, and not differentiable at $x = 0$ not big issue
− Learning doesn't happen for examples with zero activation (fixed with leaky ReLU or maxout)

Softplus unit, $\zeta(x) = log(1 + exp(x))$ with $\frac{\partial\zeta(x)}{\partial x} = \sigma(x)$ it resembles ReLU but it is differentiable everywhere so maybe it performs better, but empirically it performs worse.
Leaky ReLU/PReLU (learnable alpha), $f(x) = max(\alpha x, x)$ where $\alpha = 0.01$ and can be a hyper-parameter. The leaky ReLU avoids dead units when $x < 0$ allowing learning to continue.
Exponential Linear Unit ELU, $f(x) = max(\alpha(exp(x) - 1), x)$ this is very similar to Leaky ReLU except it requires more computation because of e^x
Swish, $\text{swish}(z) = x\sigma(\beta x) = \frac{x}{1+e^{-\beta x}}$ again similar to ReLU.
The output activation interacts with the cost function
For example let's say our output is a scalar z and if we have a large positive z we predict class 1, and when we have a large negative z we predict class 0. $\sigma(z) = Pr\{x^{(i)}\text{belongs to class1}\}$ the softmax function is the multiclass generalization of this sigmoid for binary classification.
Is it better to use **MSE** or **CE** as the cost function where $\text{MSE} = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \sigma(z^{(i)}))^2$ and $\text{CE} = -\sum_{i=1}^{n}[y^{(i)}log\sigma(z^{(i)}) + (1 - y^{(i)})log(1 - \sigma(z^{(i)}))]$
MSE as a loss function is a very poor choice because when we're far away from the answer (large negatives/positives) changing our input to the activation will not change the loss at all which is unhelpful. If we start off with a bad answer we will never change it to become a better value.
Whenever you have an output that is sigmoid or softmax we never want to use **MSE** loss we always want to use some loss like **CE**

6.2 Backpropogation

The neural network will have multiple layers with weights and biases for these layers $\mathbf{W}, \mathbf{b}$ and activation functions f in between the layers. The output of the neural network is then passed to softmax where we get our final loss $\mathcal{L}$
Our learning rule is **SGD** so $\theta \leftarrow \theta - \epsilon\frac{\partial\mathcal{L}}{\partial\theta}$ but we don't know this gradient which are $\frac{\partial\mathcal{L}}{\partial\mathbf{W}_i}$. The algorithm to calculate these gradients is called backpropogation. All it is is a fancy name for the chain rule of calculus (applies the chain rule in an operationalized fashion)
Forward propogation/forward pass: start at input and do neural network operations to calculate $\mathbf{z}$ and then $\mathcal{L}$
Backpropogation/Backward pass: start with the derivative $\frac{\partial\mathcal{L}}{\partial\mathbf{z}}$ (gradient of the loss with respect to the softmax scores) then running backwords through the network to compute $\frac{\partial\mathcal{L}}{\partial\mathbf{w}_3}$ then $\frac{\partial\mathcal{L}}{\partial\mathbf{w}_2}$, $\frac{\partial\mathcal{L}}{\partial\mathbf{w}_1}$
Why do we need backpropogation?

- Backpropogation is computationally efficient because we find most of the terms needed in backprop can be cached from forward pass
- Informally sometimes taking analytical multivariate gradients can be challenging. Backprop breaks these down into easier steps
- It is not the learning algorithm it is the method for computing gradients
- It is not specific to multi-layer NNs but a general way to compute derivatives

The **upstream gradient** is the derivative of the loss function with respect to the last wire in the network. (typically gradient of the loss with respect to the softmax classifier).
Basic step w/ chain rule, if we are looking to find $\frac{\partial\mathcal{L}}{\partial z}$ we know by the chain rule that it is $\frac{\partial f}{\partial z}\frac{\partial\mathcal{L}}{\partial f}$ which means the local gradient times the upstream gradient.
The basic intuition of backpropogation is that we break up the calculation of the gradient into **small and simple steps**. Each of these nodes in the graph is a straightforward gradient calculation where we multiply an **input** (upstream derivative/gradient) with a **local derivative/gradient** (chain rule)

6.3 Backpropogation Rules

Rules we can derive from backpropogation steps on simple examples.

- **Addition:** Just pass the upstream gradient through
- **Multiplication:** Take the upstream gradient and multiply by the value on the other wire
- **Max:** Take the upstream gradient and multiply it by the indicator function if $x > y$ where x is the wire we are finding the gradient for. e.g. $\mathbb{I}(x > y)\frac{\partial\mathcal{L}}{\partial f}$
- **Inv:** Upstream gradient times $-\frac{1}{x^2}$ where x is the value on the wire
- **Exp:** If we have a wire b which is the output of a *exp* gate with c as input e.g. $b = e^c$ the local gradient $\frac{\partial b}{\partial c} = e^c$ so we take that and multiply by the upstream
- **Log (natural):** Upstream gradient times $\frac{1}{x}$
- **Transpose:** Transpose the upstream gradient

The local gradient of a node (function) is calculated by taking the derivative of it with respect to its inputs.
What happens when two gradient paths converge? Two wires going to the same node e.g. we have a node h with two wires coming off of it $q_1 = h(x)$ and $q_2 = h(x)$ (because the input to the h node is x. What is our gradient $\frac{\partial\mathcal{L}}{\partial x}$? By law of total derivative the gradient $= \sum_{i=1}^{n}\frac{\partial q_i}{\partial x}\frac{\partial\mathcal{L}}{\partial q_i}$

7 Lecture 7

To do multivariate backpropogation, we need a multivariate chain rule. (Matrices and vectors)

7.1 Derivative of a Vector w.r.t a Vector

Let $\mathbf{y} \in \mathbb{R}^m$, what dimensionality should the derivative of $\mathbf{y}$ with respect to $\mathbf{x}$ be?

- e.g. to see how $\Delta\mathbf{x}$ modifies y_i, we would calculate $\Delta y_i = (\nabla_\mathbf{x} y_i)^T\Delta\mathbf{x}$
- This suggests that the derivative ought to be an $n \times m$ matrix, denoted $\mathbf{J} = \begin{bmatrix}(\nabla_\mathbf{x} y_1)^T\\ (\nabla_\mathbf{x} y_2)^T\\ ...\\ (\nabla_\mathbf{x} y_n)^T\end{bmatrix}$ where $(\nabla_\mathbf{x} y_i)^T = \begin{bmatrix}\frac{\partial y_i}{\partial x_1} & \frac{\partial y_i}{\partial x_2} & ... & \frac{\partial y_i}{\partial x_m}\end{bmatrix}$ This matrix would tell us how a small change in $\Delta\mathbf{x}$ results in a small change in $\Delta\mathbf{y}$ according to the formula $\Delta\mathbf{y} \approx \mathbf{J}\Delta\mathbf{x}$
- The matrix $\mathbf{J}$ is called the **Jacobian** $\in \mathbb{R}^{n\times m}$ and it tells how wiggling a vector $\Delta\mathbf{x}$ affects another vector $\Delta\mathbf{y}$

In the denominator layout definition, the denominator vector changes in a column e.g. $\frac{\partial\mathbf{y}}{\partial\mathbf{x}} = \begin{bmatrix}\frac{\partial\mathbf{y}_1}{\partial\mathbf{x}} & \frac{\partial\mathbf{y}_2}{\partial\mathbf{x}} & ... & \frac{\partial\mathbf{y}_n}{\partial\mathbf{x}}\end{bmatrix}$ and each element $\frac{\partial\mathbf{y}_i}{\partial x} = \begin{bmatrix}\frac{\partial\mathbf{y}_i}{\partial\mathbf{x}_1}\\ \frac{\partial\mathbf{y}_i}{\partial\mathbf{x}_2}\\ ...\\ \frac{\partial\mathbf{y}_i}{\partial\mathbf{x}_m}\end{bmatrix}$
Think about it as the dimension of the vector in the denominator is the first dimension of the resulting matrix
Hence the notation we use for the **Jacobian** in denominator form would be $(\nabla_\mathbf{x}\mathbf{y})^T$ (ys change in column xs change in row)
Example: $\nabla_\mathbf{x}\mathbf{y} = \mathbf{W}^T$ when $\mathbf{y} = \mathbf{W}\mathbf{x}$ because $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^h$ and $\nabla_\mathbf{x}\mathbf{y} \in \mathbb{R}^{n\times h}$
Hessian
The multivariate generalization of a second derivative. (the gradient of the gradient $\nabla_\mathbf{x}(\nabla_\mathbf{x} f(\mathbf{x}))$

7.2 Multivariate (Vector) Chain Rule

From the scalar chain rule we know if $y = f(x)$ and $z = g(y)$, then $\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial z}{\partial y}$ (intuitavely how wiggling x wiggles z by first seeing how it wiggles y then seeing how y wiggles z)

For vectors, we have $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{R}^p$ with $\mathbf{y} = f(\mathbf{x})$ and $\mathbf{z} = g(\mathbf{y})$

- $\Delta\mathbf{x} \to \Delta\mathbf{z}$ (how wiggling $\mathbf{x}$ wiggles $\mathbf{z}$) $\Delta\mathbf{z} \approx (\frac{\partial \mathbf{z}}{\partial \mathbf{x}})^T \Delta\mathbf{x} \to (p) \approx (p \times m)(m)$
- $\Delta\mathbf{x} \to \Delta\mathbf{y} \to \Delta\mathbf{z}$ (how wiggling $\mathbf{x}$ wiggles $\mathbf{y}$ wiggles $\mathbf{z}$) $\Delta\mathbf{y} \approx (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T \Delta\mathbf{x} \to (n) \approx (n \times m)(m)$ and $\Delta\mathbf{z} \approx (\frac{\partial \mathbf{z}}{\partial \mathbf{y}})^T \Delta\mathbf{y} \to (p) \approx (p \times n)(n) = (\frac{\partial \mathbf{z}}{\partial \mathbf{y}})^T (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T \Delta\mathbf{x}$

Tensor Derivatives

$\mathbf{y} = \mathbf{W}\mathbf{x}$ in a NN this corresponds to $\mathbf{h_2} = f(\mathbf{W_2 h_1})$ in this example $\mathbf{y} \in \mathbb{R}^m$, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{W} \in \mathbb{R}^{m\times n}$
$\frac{\partial \mathbf{y}}{\partial \mathbf{W}} \in \mathbb{R}^{m\times n\times m}$ remember $\frac{\partial \mathbf{y_1}}{\partial \mathbf{W}} \in \mathbb{R}^{m\times n}$ and we will just stack $\forall_{i=1}^m \frac{\partial \mathbf{y}_i}{\partial \mathbf{W}}$ matrices resulting in a tensor $\in \mathbb{R}^{m\times n\times m}$

Example: $\frac{\partial \mathbf{z}}{\partial \mathbf{W}} \in \mathbb{R}^{m\times n\times m}$ where $\mathbf{z} \in \mathbb{R}^m$, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{z} = \mathbf{y} - \mathbf{W}\mathbf{x}$
$\frac{\partial \mathbf{z}_k}{\partial \mathbf{W}} \in \mathbb{R}^{m\times n}$ (there will be m of these $k = \{1, ..., m\}$ it will have 0s in every single row except the kth row where the row will be $\begin{bmatrix} -\mathbf{x}_1 & -\mathbf{x}_2 & \dots & -\mathbf{x}_n \end{bmatrix}$

A few notes on tensor derivatives

- In general, the simpler rule can be inferred via pattern intuition / looking at the dimensionality of the matrices, and these tensor derivatives need not be explicitly derived
- Actually calculating them and storing them is usually not a good idea for memory and computation reasons
- If we know the end result is a simple outer product of two vectors, we need not even calculate an additional derivative in this step of backpropogation (assuming inputs were cached)

Intuition, since we know $\nabla_{\mathbf{x}}\mathbf{W}\mathbf{x} = \mathbf{W}^T$ then $\nabla_{\mathbf{W}}\mathbf{W}\mathbf{x}$ ought to look like $\mathbf{x}^T$. We know that $\frac{\partial \epsilon}{\partial \mathbf{z}} \in \mathbb{R}^m$ and $\frac{\partial \epsilon}{\partial \mathbf{W}} \in \mathbb{R}^{m\times n}$. By the chain rule $\frac{\partial \epsilon}{\partial \mathbf{W}} = \frac{\partial \mathbf{z}}{\partial \mathbf{W}}\frac{\partial \epsilon}{\partial \mathbf{z}}$ and because $\frac{\partial \epsilon}{\partial \mathbf{W}} \in \mathbb{R}^{m\times n}$ and $\frac{\partial \epsilon}{\partial \mathbf{z}} \in \mathbb{R}^{m\times 1}$ if we said that instead of $\frac{\partial \mathbf{z}}{\partial \mathbf{W}}$ being a tensor it looked like $\mathbf{x}^T \in \mathbb{R}^{1\times n}$ and we multiplied it on the other side for an outer product we would get the correct answer, meaning $\frac{\partial \epsilon}{\partial \mathbf{W}} = \frac{\partial \epsilon}{\partial \mathbf{z}}\mathbf{x}^T$

THIS TRICK WORKS WITH MATRICES TIMES MATRICES TOO

Example with backprop: $\mathbf{y} = \mathbf{W}\mathbf{x}$ then $\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \mathbf{W}^T\frac{\partial \mathcal{L}}{\partial \mathbf{y}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}}\mathbf{x}^T$

Any operation in the computational graph that is applied element wise (like ReLU) will result in a hadamard product. e.g. If we have an input to a ReLU $\mathbf{x}$ and its outputs $\mathbf{h}_2$ (both vectors) the backprop $\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \mathbb{I}\{\mathbf{x} > 0\} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{h_2}}$

7.3 Initializations

The initializations of weights in biases for each neural network layer have an impact on the model. If for example we set them all as **small random weights** $w_{i,j} \leftarrow N(0, 0.01)$ where $N(0, 0.01)$ means a normal distribution with 0 mean and 0.01 variance. **Xavier Initialization** is $w_{i,j} \leftarrow N(0, \frac{1}{n})$ where n is the number of neurons in the prior layer.

It is bad for learning if the output activations are close to zero because $\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = 0$ so no learning occurs.

8 Lecture 8

8.1 Initializations Continued

Large random weight initialization: $w_{i,j} \leftarrow N(0, 1)$ our activations will explode with the means and standard deviations of the activations being very high at later hidden layers. This is bad because in gradient descent we will be taking large gradient steps and our gradient will not converge.

If not small and not large initialization, then what?

8.2 Xavier initialization

We want the variances of the activations in layer 1 to equal the variances of the activations in layer 2 ... to the last layer. (keep the variance of the units similar between each layer).

For simplicity, we assume the input has equal variance across all dimensions. Then, each unit in each layer ought to have the same statistics. h_i denotes a unit in the ith layer, but the variances ought to be the same for all units in this layer.

Concretely $\text{var}(h_i) = \text{var}(h_j)$ and $\text{var}(\nabla_{h_i}\mathbf{J}) = \text{var}(\nabla_{h_j}\mathbf{J})$. We are going to assume that each artificial neuron in one layer has the same statistics as eachother (same expected value and variance etc). How do we set the weights initially so that the variance of the neurons in every layer are equal.

Assuming the units are linear, then $h_i = \sum_{j=1}^{n_{in}} w_{i,j}h_{i-1,j}$ where each layer has n_{in} neurons. Further if $w_{i,j}$ and h_{i-1} are independent, and all the units in the $(i-1)$th layer have the same statistics, using the fact $\text{var}(wh) = \mathbb{E}^2(w)\text{var}(h) + \mathbb{E}^2(h)\text{var}(w) + \text{var}(w)\text{var}(h)$ which is the same as $= \text{var}(w)\text{var}(h)$ if $\mathbb{E}(w) = \mathbb{E}(h) = 0$ then $\text{var}(h_i) = \text{var}(h_{i-1}) \cdot \sum_{j=1}^{n_{in}} \text{var}(w_{i,j})$ so we want this **sum** to be 1

Because we are assuming the statistics are shared between all activations then $n_{in} \times \text{var}(w) = 1$ solving for variance we get $\text{var}(w) = \frac{1}{n_{in}}$ **Which is what we will initialize the variance of our weights to be so our activations don't explode or become 0**

Since we also wanted the gradients to have similar variances we get for each connection j in layer i $\text{var}(w_{i,j}) = \frac{1}{n_{out}}$ so we incorporate both of these in the Xavier initialization by setting the weight distribution to $N(0, \frac{2}{n_{in}+n_{out}})$ **but oftentimes the nout is left out**

Important Note: If we use this Xavier initialization $\frac{1}{n_{in}}$ with ReLU lots of the units will go to 0 instead. If we assume that ReLU will kill off 50% of our units so we should increase the variance by a factor of 2, motivating $\frac{2}{n_{in}}$. Eventually it is suggested to use a uniform distribution $U(-\frac{\sqrt{6}}{n_{in}+n_{out}}, \frac{\sqrt{6}}{n_{in}+n_{out}})$

8.3 Batch Normalization

Will help neural network be far less sensitive to hyperparameter settings and initializations

Motivated through internal covariate shift, but also works by making the loss surface smoother

Take a neural network where we have multiple layers, say we are trying to update the weights of the third layer w_3 we do this through the update step $w_3 \leftarrow w_3 - \epsilon\nabla_{\mathcal{L}}w_3$ the problem is we are only updating it by looking at how wiggling w_3 will change the loss but the loss can also be changed by other things not just w_3 and at every gradient descent step we are updating all weights meaning the gradients which we had calculated based on the output of a previous layer are now stale because we updated the previous layer with gradient descent as well. (this is dangerous because we update the weights in parallel)

This can be avoided by having some vector $\tilde{w}$ which has all weights in it and we then take $\nabla_{\mathcal{L}}\tilde{w}$ because we will be seeing how wiggling all weights will effect the loss.

The idea of batch normalization is that we know when w_3 is changing it expected h_2 to have the statistics from the last training batch (not the statistics after update). We mitigate this by normalizing all the statistics of the hidden unit activation to be 0 mean unit variance.

$h_i = \text{relu}(x_i)$ where x_i is the input (usually $W_i h_{i-1}$) then we want to normalize $\mathbb{E}[x_i] = 0$ and $\text{var}[x_i] = 1$

So our NN will look like **affine→BN→ReLU→affine**... so we apply the batch norm before the ReLU

When we have **batch normalization** we don't need **Xavier initialization** (because of BN needing $\text{var}[x_i] = 1$) but it likely still helps

Batch-Norm can be thought of as z-scoring

For a batch of m training examples of activations (before the ReLU) we can compute the mean. $\mu_i = \frac{1}{m}\sum_{j=1}^m x_i^{(j)}$ and the standard deviation $\sigma_i^2 = \frac{1}{m}\sum_{j=1}^m (x_i^{(j)}\mu_i)^2$ after computing these we make it 0 mean by subtracting the mean and unit variance by dividing the stdev (plus some small ϵ to ensure no divide by 0)

$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$, where $\hat{x}_i$ is our normalized activation. The output of the batch norm layer will be a $y_i = \gamma_i\hat{x}_i + \beta_i$ with γ_i and β_i being learnable parameters allowing $\mathbb{E}[y_i] = \beta_i$ and $\text{var}[y_i] = \gamma_i^2$. (shift and scale parameters)

Why did we go through the trouble of getting 0 mean unit variance if we can just change it with parameters? If we restrict it that much we may overly constrain the network which may not allow the lowest loss.

Layer Normalization is the same as batch-normalization but taking m as the number of activations in the layer not just some batch.

Group Normalization is normalizing over groups in the layer so if the first 10 activations are a group we could normalize over those then the next 10 etc.

Back propogation for batch normalization (for one unit i and one example j) there are m total examples

The batch normalization layer is typically placed right before the nonlinear activation. Hence, a layer of a neural network may look like $\mathbf{h}_i = f(\text{batch-norm}(\mathbf{W}_i\mathbf{h}_{i-1} + \mathbf{b}_i))$

Empirically, batch normalization allows higher learning rates to be used (with no batch-norm you typically need lower learning rates) this is because **batch-norm** is making the loss surface smoother so the likelihood if computing a large gradient is rarer so you can have the learning rate set higher.

9 Lecture 9

9.1 Regularizations

Regularizations are used to improve model generalization. **[Regularization is] any modification we make to a learning algorithm that is intended to reduce its generalization error but**

not its training error

- Tends to increase the estimator bias while reducing the estimator variance
- Can be seen as a way of preventing overfitting
- Common problem is picking the model size and complexity, it may be fine to choose a large model with good regularization

Early Stopping
Constantly evaluate the training and validation loss on each iteration and return the model with the lowest validation error.

9.2 Norm Penalties

A common and simple type of regularization is to modify the cost function with a parameter norm penalty. This penalty is typically denoted as $\Omega(\theta)$ and results in a new cost function of the form $\mathcal{L}(\theta; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\theta)$ where $\alpha \geq 0$ and it dictates how much you care about the parameter norm penalty over the original loss (hyperparameter, can strongly affect generalization).

L^2 regularization
Penalize the size of weights, also called *ridge regression* this promotes models with parameters that are closer to 0 (simpler). $\Omega(\theta) = \frac{1}{2}\mathbf{w}^T\mathbf{w} = \frac{1}{2}\|\mathbf{w}\|^2$. Intuitively, to prevent $\Omega(\theta)$ from getting large L^2 regularization will cause weights $\mathbf{w}$ to have small norm.
The new cost function is $\tilde{\mathcal{L}}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \mathcal{L} + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$ with the corresponding gradient $\nabla_\mathbf{w}\tilde{\mathcal{L}} = \alpha\mathbf{w} + \nabla_\mathbf{w}\mathcal{L}$ so our update step now becomes $\mathbf{w} \leftarrow (1 - \epsilon\alpha)\mathbf{w} - \epsilon\nabla_\mathbf{w}\mathcal{L}$ formalizing the shrinking of weights (weight decay)
Why we want smaller weights example: Say we want $y = 1$ with $x_1 = 0.1$ and $x_2 = 0.01$ to make 1 we can do $100x_1 - 900x_2$ but now if we have small deviations in the inputs say $x_1 = 0.11$ our prediction would now be 2 **when we have large weights we're more sensitive to fluctuations in the inputs**
Extensions of L^2 regularization
Instead of a soft constraint that $\mathbf{w}$ be small, one may have prior knowledge that $\mathbf{w}$ is close to some value $\mathbf{b}$. Then the regularizer may take the form: $\Omega(\theta) = \|\mathbf{w} - \mathbf{b}\|_2^2$ or if we want $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$ to be close we can have $\Omega(\theta) = \|\mathbf{w}^{(1)} - \mathbf{w}^{(2)}\|_2^2$
L^1 Regularization instead of penalizing the 2-norm we penalize the 1-norm
$\Omega(\theta) = \|\mathbf{w}\|_1 = \sum_i |w_i|$ Intuitively this penalty also causes the weights to be small. However, because the subgradient of $\|\mathbf{w}\|_1$ is sign($\mathbf{w}$) the gradient is the same regardless of the size of $\mathbf{w}$ (think of the graph of the absolute value). The gradients of the 1-norm are always the same. Empirically if we penalize the weights with the 1-norm the weights will go to 0

- This may be useful for feature selection, because features with 0 weights are discarded (because they weren't useful for predicting the output)

Pruning the network do an L^1 regularization and remove the features with weights 0 (this leads to smaller models [more memory saving])
Sparse Representations instead of having sparse parameters (e.g. elements of $\mathbf{w}$ being sparse) we may want to have sparse representations. Imagine a hidden layer of activity $\mathbf{h}^{(i)}$ to achieve this we may set $\Omega(\mathbf{h}^{(i)}) = \|\mathbf{h}^{(i)}\|_1$

9.3 Dataset Augmentation

We can flip the image, crop it, adjust the brightness, lens correction, rotate. By doing these dataset augmentations we increase the size of our training data and increase the robustness of the classification for those images.
Label Smoothing add noise to network by saying our labels are noisy. Instead of having a *one hot* representation of the label for a given image (a vector with all 0s for each class the image is not and a single 1 for the index of class the image is part of) we can have label smoothing where each label index has a probability (so instead of a single 1 we have something like 0.9 for the correct class and distribute the remaining 0.1 across all other class indexes).
Cross Entropy Loss for one hot $\mathcal{L}^{(i)} = \sum_{c=1}^{C} -y_c log(p_c)$ where y_c is either 0 or 1 (if correct class), and p_c is the probability of the model for the class.
Cross Entropy Loss for label smoothing $\mathcal{L}^{(i)} = -y_c(1-\alpha) * log(p_c) + \frac{\alpha}{C-1}$ where $y_c \in \{0, 1\}$, $\alpha = 0.1$ (hyperparameter) (probability for the incorrect class), C is the number of classes and $-y_c(1-\alpha) * log(p_c)$ is the negative log likelihood for the correct class. (the $\frac{\alpha}{C-1}$ should be summed for all incorrect classes)
Multitask Learning If we have one task like image-net classification it may lead to features specific to the classification problem but in general is looking at the image and determining what it is and a lot of the features used for this may be used for other classification tasks. (example tasks below)

- **Semantic Decoding:** Labeling the class of every pixel
- **Instance Decoding:** Labeling all the instance of one class
- **Depth Decoding:** Seeing how far something is in a given image

We can say that the output of a neural network must use these tasks. For each of these tasks (decodings) we have a loss and the multi-task loss is a sum or weighted sum of these. When we compute the gradients we back propogate through the entire network, now each of these tasks share the same features from the NN (upstream) this has the regularizing effect of helping the features be more general.
Transfer Learning (good to prevent overfitting for tasks with little data)
The idea is that if tasks are similar enough, then the features at later layers of the network ought to be good features for the new task. If little training data is available but the tasks are similar you just train a new linear layer at the output of the pre-trained network. Even if more data is available for the task it may still be good to use transfer learning.
We freeze the trained network from the similar task and just train the new linear layer at the output with the limited data.

9.4 Ensembling

Approach is to train multiple different models and average their results together at test time. (because the models might make independent errors) **This almost always increases performance by substantial amounts**
Intuition is that if they are independent models they will make independent errors
With k independent models, the average model error will decrease by a factor of $\frac{1}{k}$. Denoting ϵ_i to be the error of model i on an example and assuming $\mathbb{E}\epsilon_i = 0$ means the statistics of the error is the same across all models.
$\mathbb{E}[(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i)^2] = \frac{1}{k}\mathbb{E}\epsilon_i^2$ (if the models make independent errors)
$\frac{1}{k}\mathbb{E}\epsilon_i^2 + \frac{k-1}{k}\mathbb{E}[\epsilon_i\epsilon_j]$ (if the models are not independent) worst case they make the same mistakes $\epsilon_i = \epsilon_j$
Bagging (bootstrap aggregating)
The way we make k models to average is by making k datasets where we draw N examples from the training set with replacement, we do this for k datasets.

- Construct k datasets using the bootstrap (Data size N and draw with replacement to get N samples, do this k times e.g. $k = 2$ $N = \{1, 2, 3, 4\}$ we could have dataset $1 = \{1, 2, 1, 3\}$ and dataset $2 = \{4, 3, 2, 3\}$)
- Train k models on these k datasets

In practice, neural networks reach a wide variety of solutions with different initializations, hyperparameters etc meaning even if trained from same dataset they tend to produce partially independent errors. (k models with different inits where we train on the same dataset)
Model averaging is very expensive

9.5 Dropout

The way we get the benefit of ensembling in neural networks. Dropout is a computationally inexpensive yet effective method for generalization. It can be viewed as **approximating the bagging procedure** for exponentially many models, while only optimizing a single set of parameters.

- On a given training iteration sample a binary mask (each element $\in \{0, 1\}$) where 1 occurs w.p p and 0 w.p $1 - p$
- One hyperparameter per layer p which is the probability of keeping a neuron in a layer, typical values $p = 0.8$ for inputs and $p = 0.5$ for hidden units
- Apply the mask to all units (hadamard product) then perform forward and backward pass and parameter update
- In *prediction*, multiply each hidden unit by the parameter of its Bernboulli mask, p

If we have a NN with N units there are 2^N possible masks, in every epoch we change the mask we use for dropout.
What about test time? e.g. Training iteration 1 $M = [1, 0, 1, 0] \Rightarrow h_{out} = \text{relu}(w_1h_1 + w_3h_3)$
Training iteration 1 $M = [0, 1, 0, 1] \Rightarrow h_{out} = \text{relu}(w_2h_2 + w_4h_4)$
Naively if we said $h_{out} = \text{relu}(w_1h_1 + w_2h_2 + w_3h_3 + w_4h_4)$ but the statistics of this h_{out} don't match the statistics of the training. To make them correct we scale by p e.g. $h_{out} = \text{relu}(w_1h_1 + w_2h_2 + w_3h_3 + w_4h_4) * p$ this is called the **weight scaling inference rule**

9.6 Inverted Dropout

With regular **Dropout** we need our p in testing but we don't want that. Instead we use **Inverted Dropout**. When we generate the mask it's no longer just 1s and 0s but 1s and 0s divided by p (this can be done either to the mask or the activations). This will scale the activations in training so that in test time we don't have to multiply by p.
How is this a good idea?

1. Dropout approximates bagging since each mask is basically a different model
2. Think of the dropout as regularizing each hidden unit to work well in different contexts (masks)
3. Dropout may cause units to encode redundant features (multiple neurons encode necessary features for classification)

10 Lecture 10

10.1 Optimization for Neural Networks

Can we improve the SGD algorithm?

Vanilla GD: the gradient descent step is $\theta \leftarrow \theta - \epsilon\nabla_\theta\mathcal{L}(\theta)$ where $\mathcal{L}$ is the loss function, θ is the parameters, and ϵ is a **hyperparameter** for the step size.

Stochastic GD: Set a learning rate ϵ and an initial parameter setting θ. Set a minibatch size of m examples.

- Sample m examples from the training set $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)}\}$ and their outputs $\{\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, ..., \mathbf{y}^{(m)}\}$
- Compute the gradient estimate $\mathbf{g} = \frac{1}{m}\sum_{i=1}^{m}\nabla_\theta\mathcal{L}(\theta)$ and update $\theta \leftarrow \theta - \epsilon\nabla_\theta\mathcal{L}(\theta)$

10.2 Momentum

In momentum we maintain the running mean of the gradients which then updates the parameters (we will step along the running mean).

- Initialize $\mathbf{v} = 0$. Set $\alpha \in [0, 1]$ (typical values are 0.9 or 0.99)
- Until criterion is met compute gradient $\mathbf{g}$ update $\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\mathbf{g}$ and gradient step $\theta \leftarrow \theta + \mathbf{v}$

A reason why we like momentum is because of zig-zagging gradients. If the gradients zig-zag the running mean will point in the direction to cancel out the zig-zag.

Example: With steps $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$ we have the following intermediate momentums. $\mathbf{v} = 0$, $\mathbf{v}_1 = -\epsilon\mathbf{g}_1$, $\mathbf{v}_2 = -\epsilon(\alpha\mathbf{g}_1 + \mathbf{g}_2)$ and $\mathbf{v}_3 = -\epsilon(\alpha^2\mathbf{g}_1 + \alpha\mathbf{g}_2 + \mathbf{g}_3)$

Momentum can find different local minimums but will converge faster and tend to be better

+ If there are steep local minima then the gradient with momentum can step over them (and it is generally the case that these are poor minimum)
− If the steep local minima is the best minima then it is likely to take longer to get to the bottom

Does momentum help with local optima? (e.g. can we escape bad local optima) YES

Momentum tends to find shallow/flat local optima shallow and flat optima are better because they are less sensitive to the settings of the parameters. If our θ changes just a bit the loss doesn't change as much as if we were in a deep narrow minima.

10.3 Nesterov Momentum

Instead of computing the gradient before stepping in the direction of momentum we want to compute the gradient from where we would end up after stepping in the direction $\alpha\mathbf{v}$ because we will always step there.

- **Update:** $\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_\theta\mathcal{L}(\theta\alpha\mathbf{v})$ because $\theta + \alpha\mathbf{v}$ is where we would be after stepping with momentum
- **Gradient Step:** $\theta \leftarrow \theta + \mathbf{v}$

This is done by performing a change of variables with $\tilde{\theta}_{old} = \theta_{old} + \alpha\mathbf{v}_{old}$ and this doesn't require evaluating the gradient at $\theta + \alpha\mathbf{v}$

- **Update:** $\mathbf{v}_{new} = \alpha\mathbf{v}_{old} - \epsilon\nabla_{\tilde{\theta}_{old}}\mathcal{L}(\tilde{\theta}_{old})$
- **Gradient Step:** $\tilde{\theta}_{new} = \tilde{\theta}_{old} + \mathbf{v}_{new} + \alpha(\mathbf{v}_{new} - \mathbf{v}_{old})$
- **Set:** $\mathbf{v}_{new} = \mathbf{v}_{old}, \tilde{\theta}_{new} = \tilde{\theta}_{old}$

Nesterov Momentum will correct faster than regular momentum because it computes the gradient after the step.

Techniques to adapt the learning rate: Choosing ϵ judiciously can be important for learning. In the beginning a larger learning rate is typically better since bigger updates may accelerate the learning. However it may need to slow down as time goes on. **We often apply a decay rule to the learning rate called annealing**. One way to do this is updating based on history of gradients.

10.4 Adaptive Gradient (Adagrad)

Learning rate is decreased through division by historical gradient norms. $\mathbf{a}$ denotes a running sum of squares of gradient norms. Initialize $\mathbf{a} = 0$ and set ν to a small value (to avoid zero division)

- **Compute the gradient:** $\mathbf{g}$ and update $\mathbf{a} \leftarrow \mathbf{a} + \mathbf{g} \odot \mathbf{g}$
- **Gradient Step:** $\theta \rightarrow \theta - \frac{\epsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$

The more we have stepped in a dimension (the larger its a will be) the less we will step in that dimension in the future

Is there a problem with Adagrad? $a_1 = a_1 + g_1^2$ our values of a will never get smaller

10.5 RMSProp

Augmentation to Adagrad by making the gradient accumulator $\mathbf{a}$ an exponentially weighted moving average. Initialize $\mathbf{a} = 0$ and set ν to be small. Set β to be between 0 and 1 (0.99)

- **Compute Gradient:** $\mathbf{g}$ and update $\mathbf{a} \leftarrow \beta\mathbf{a} + (1 - \beta)\mathbf{g} \odot \mathbf{g}$
- **Gradient Step:** $\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$

Allows us to step big in the direction we have stepped before, because **a**s aren't strictly decreasing

---

RMSProp w/ momentum

RMSProp can be combined with momentum as follows, intitialize $\mathbf{a} = 0$, set α, β to be between 0 and 1 set $\nu = 1e - 7$

- **Compute Gradient:** $\mathbf{g}$, accumulate gradient $\mathbf{a} \leftarrow \beta\mathbf{a} + (1 - \beta)\mathbf{g} \odot \mathbf{g}$
- **Momentum:** $\mathbf{v} \leftarrow \alpha\mathbf{v} - \frac{\epsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$ **Gradient Step:** $\theta \leftarrow \theta + \mathbf{v}$

10.6 Adam

Most commonly used optimizer, composed of momentum-like step followed by Adagrad/RMSProp like step.

Adam w/ no bias correction Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment". Set β_1 and β_2 to between 0 and 1. (good $\beta_1 = 0.9$ and $\beta_2 = 0.999$) initialize ν to be small.

- **Compute Gradient:** $\mathbf{g}$, first moment update $\mathbf{v} \leftarrow \beta_1\mathbf{v} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$
- **Second Moment Update:** $\mathbf{a} \leftarrow \beta_2\mathbf{a} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$ and **gradient step** $\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{v}$

Adam w/ bias correction Bias correction means when we are taking our $\mathbf{v}$ and $\mathbf{a}$ they are running means but at the start there are only a few samples so the average is inaccurate. (Helps to get more accurate estimates of momentum and second moment)

Keep everything the same from **Adam w/o bias correction** but include a $t = 0$. Same steps with the following modifications

- **Time Update:** after gradient update $t \leftarrow t + 1$
- **We Perform Bias correction in moments:** $\tilde{\mathbf{v}} = \frac{1}{1 - \beta_1^t}\mathbf{v}$ and $\tilde{\mathbf{a}} = \frac{1}{1 - \beta_2^t}\mathbf{a}$
- **Gradient Step:** $\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{\tilde{\mathbf{a}}} + \nu} \odot \tilde{\mathbf{v}}$

Has 3 hyperparameters (β_1, β_2, and ϵ)

All discussed optimizers are first order methods (we only use first derivative)